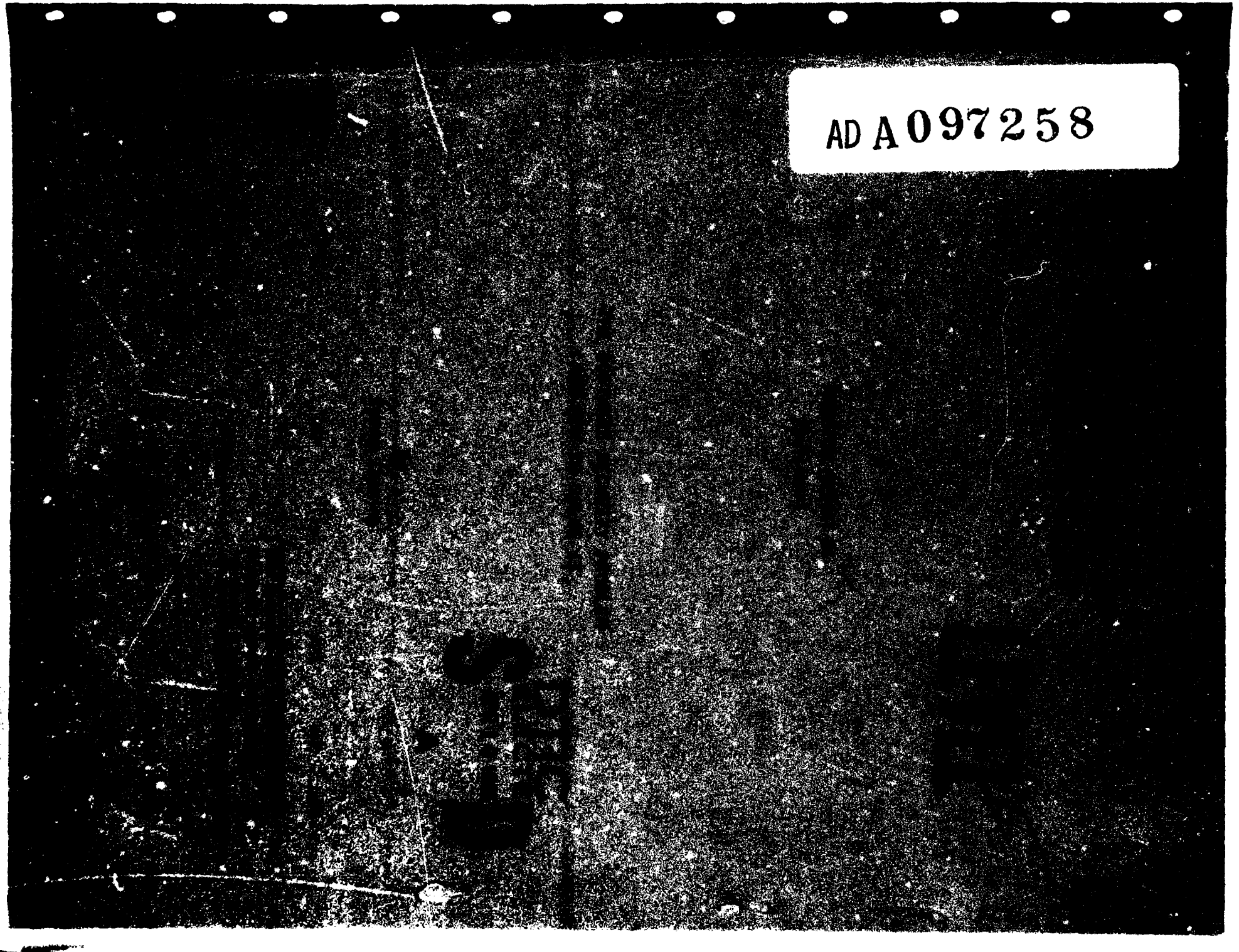

AD A097258

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER: TR-6 | 2. GOVT ACCESSION NO.: AD-A097258 |
| 3. RECIPIENT'S CATALOG NUMBER: | |
| 4. TITLE (and Subtitle): THE SOFTWARE SECURITY PROBLEM AND HOW TO SOLVE IT | 5. TYPE OF REPORT & PERIOD COVERED: Final Report SEP 1976 - JUL 1977 |
| | 6. PERFORMING ORG. REPORT NUMBER: |
| 7. AUTHOR(s): Steven Cushing | 8. CONTRACT OR GRANT NUMBER(s): DAAG29-76-C-0061 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: Higher Order Software, Inc. 806 Massachusetts Avenue Cambridge, MA 02139 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: P-14612-A-M |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Department of the Army Army Research Office Research Triangle Park, NC | 12. REPORT DATE: July 1977 |
| | 13. NUMBER OF PAGES: 70 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | 15. SECURITY CLASS. (of this report): Unclassified |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

security, protection, specification, Higher Order Software, hierarchical modeling

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

There are many similarities between the requirements of security and the requirements of reliability. The Higher Order Software (HOS) methodology was developed as a means of guaranteeing system reliability and as such is an inherently secure system. HOS manages to solve these two problems of security and reliability by showing that they need not arise in the first place. If software is specified according to the principles of HOS, then there is no need to ask how to prevent data or timing conflicts, because there simply will be no such thing. Similarly, if software had

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

always been specified according to HOS, then it would never have occured to anyone to ask how to make a software system secure because it simply would have been secure already.

HIGHER ORDER SOFTWARE, INC.
806 Massachusetts Avenue
Cambridge, MA 02139

(14) HOS-TR-6-R1

TECHNICAL REPORT #6
Revision 1

# THE SOFTWARE SECURITY PROBLEM AND HOW TO SOLVE IT. Revision 1.

by
Steven Cushing



July 1977

## ACKNOWLEDGEMENTS

This report was prepared under Contract No. DAAG29-76-C-0061, sponsored by the Department of the Army, Army Research Office, Research Triangle Park, North Carolina.

The author would like to thank Margaret Hamilton and Saydean Zeldin for introducing me to the Higher Order Software methodology and for carefully reading earlier versions of this report and making helpful suggestions on how to improve it. Many lengthy discussions with Bill Heath have also been indispensable in helping to develop my understanding of HOS. I would also like to thank Marty Wolfe for a number of insightful comments that have been incorporated into the text and Andrea Davis for preparing the report and maintaining a cheerful demeanor through numerous changes and iterations.

The publication of this report does not constitute approval by the Army Research Office of the findings or the conclusions contained herein. It is published only for the exchange and stimulation of ideas.

S. Cushing

TABLE OF CONTENTS



FIGURES

# THE SOFTWARE SECURITY PROBLEM AND HOW TO SOLVE IT

*So if a man's wit be wandering,*
*let him study the mathematics*

*- Francis Bacon*

## 1. SECURITY AND RELIABILITY

When digital computers first began being used in the 1950's, people just programmed their computers in machine or assembly language and ran their programs. With the introduction of higher-order languages, however, and particularly with the development of large and very large software systems, such as those of the Apollo project, for example, a whole new set of questions and problems arose that the early programmers could never have imagined. How can we prevent timing conflicts? How can we prevent data conflicts? How can we prove programs correct? What is the relation between synchronous and asynchronous processing? How can we make an operating system secure? All of these questions and others constitute what Parnas [Par72a] has termed "the so-called 'software engineering' problem" (p. 330).

One of the most interesting instances of this software-engineering problem is that of guaranteeing system security. How can access to the various components of a system be restricted specifically to those for whom it is intended? Linden [Lind76] points out that there are many similarities between the requirements of security and the requirements of reliability, suggesting that "a technical breakthrough on both the security and software reliability problems appears to be as feasible as a breakthrough on the security problem alone" (p. 410). Guaranteeing security requires that "operating systems must be structured so that interactions between system modules are more clearly defined and more closely controlled" (p. 411), but "this same control over the interaction of modules is also needed for reliability."



HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

Similarly, "the protection mechanisms needed for security can also be used to enforce software modularity," and "such modularity would improve the reliability and correctness of software." In a word, "there is enough overlap between the requirements for security and the requirements for high system availability that it is reasonable to attempt to solve both problems at the same time." (Availability is a necessary part of reliability, for Linden.)

In this report we will argue that Linden is correct, by showing that software specified according to the Higher Order Software (HOS) methodology of Hamilton and Zeldin [Ham76a,b,77] is automatically secure. HOS was developed as a means of guaranteeing system reliability, without any concern for the security problem per se. Systems specified in HOS are guaranteed against ever having timing or data conflicts [Ham76b]. The fact that they also turn out to be secure makes HOS exactly the kind of common breakthrough that Linden suggests is feasible.

HOS manages to solve these two problems by showing that they need not arise in the first place. If software is specified according to the principles of HOS, then there is no need to ask how to prevent data or timing conflicts, because there simply will be no such thing. Similarly, ignoring history for the moment, if software had always been specified according to HOS, then it would never have occurred to anyone to ask how to make a software system secure, because it simply would have been secure already. Demonstrating this latter point is the purpose of our present paper.

Many people have recognized that the key to solving these problems is to make a clean separation between the specification of a system and its implementation, and, as we will see, HOS is a systems theory that really manages to do this successfully. We will see that trying to solve the reliability, security, and related problems entirely in terms of implementation is like trying to get to the moon on a skateboard. Some systems theories



HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

enable us to get off the ground, but then we are stranded forever in the orbit of implementation. HOS enables us, finally, to achieve escape velocity, break free of this orbit, and reach whatever destination we have decided on.

## 2. THE SECURITY PROBLEM

Linden [Lind76] presents a general abstract characterization of system security in terms of what he calls a protection model. Such a model "views the computer as a set of active entities called subjects and a set of passive entities called objects. The protection model defines the access rights of each subject to each object" (p. 415).

Linden represents a protection model in the form of a protection matrix, such as the one in Figure 1 [Lind76, p.416]. The rows of a protection matrix are associated with the subjects of the model and its columns are associated with the objects. "For each subject/object pair, the corresponding entry in the matrix defines the set of access rights that the subject has to the object." For the protection model represented by the protection matrix in Figure 1, for example, we see that subject C may read or execute object X, because both "READ" and "EXECUTE" appear in the matrix slot that occurs at the intersection of row C and column X.

Changes to the protection matrix itself are also controlled by the access rights represented in the matrix; "for example, a subject with 'delete' access to an object can eliminate that object from the protection matrix." Subjects can be allowed to have access rights to each other by having subjects appear also as objects in the protection matrix. "For example, one subject may be allowed to transfer control to another subject by using an 'enter' access right to the other subject."

Linden also introduces the notion of a protection environment, which includes "everything that a subject might cause to be done on its behalf by another subject," as well as everything the subject is allowed to do directly. "A protection domain is a more restricted concept and includes only access rights to objects that are accessible by the subject." The rows of the

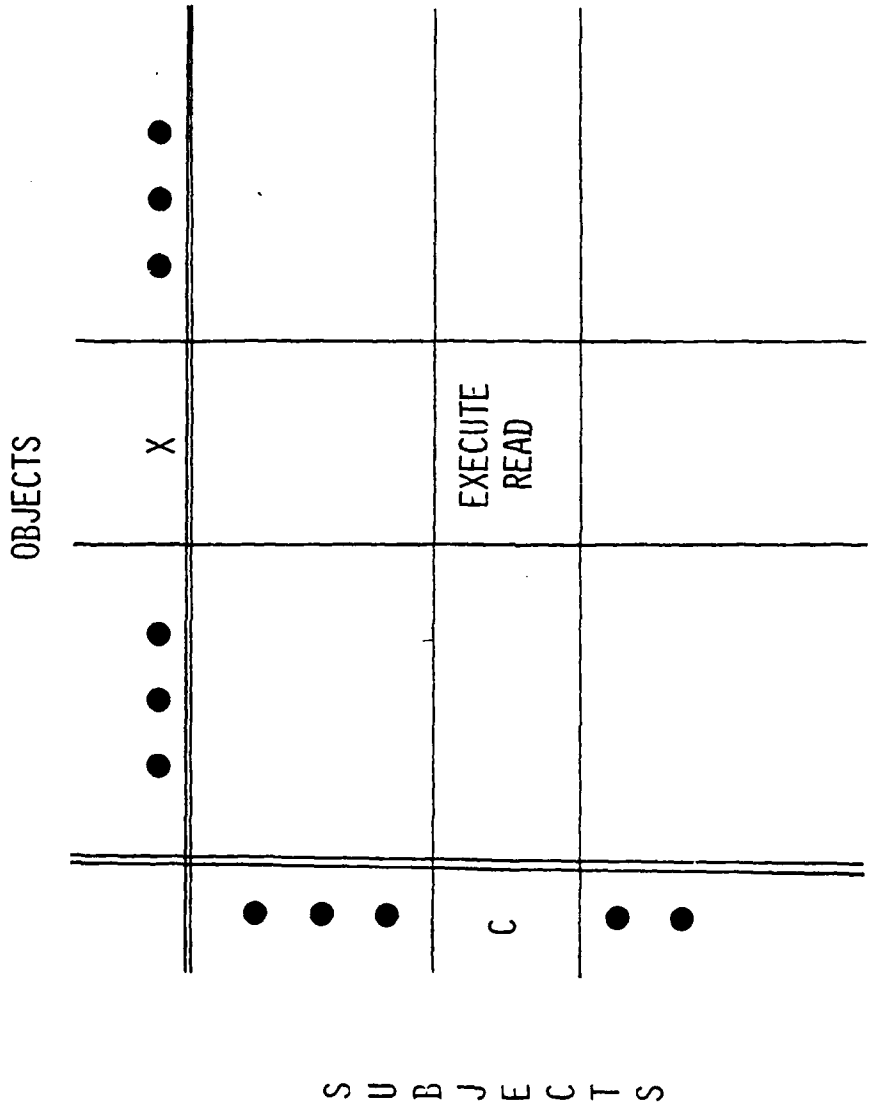



Figure 1: A Protection Matrix [Lind76, p. 416]

protection matrix represent the protection domains of the protection model.

The key to Linden's approach to system security is the notion of small protection domains. Linden uses the term "small protection domains" as "a qualitative description of a certain class of protection models. The word 'small' is not intended in a rigid quantitative sense" (p. 416). A small protection domain, for Linden, is the minimal protection domain that will still allow its subject access to everything it has to access. A protection domain may be very large in a quantitative sense, but it is a "small" protection domain if it could not be decreased in size without overly restricting the access rights of its subject. Linden calls this the "principle of least privilege."

Since "a large program usually needs access to many objects," it follows that "protection domains can be kept small only if a large program executes in many different protection domains and constantly switches between these protection domains during its execution." Protection domains can be kept small, "if small subunits of a program execute in their own protection domains," because "a small subunit of a program typically only needs access to a small number of objects." It follows that "the flexibility, ease, and efficiency of domain switching is the primary factor in determining whether protection domains can be kept small and closely tailored to actual needs."

Linden integrates protection domain switching with the calling of a procedure. This permits each procedure to have its own protection domain, even though a domain switch might not be involved in every procedure. A protected procedure, for Linden, is a procedure that does involve a domain switch.

If a procedure is a protected procedure, then it will have a particular protection domain associated with it. "Thus the

right to access certain objects may be available during the execution of that procedure--and possibly only during executions of that procedure." Each execution of a protected procedure will possess the access rights of the procedure, whatever the calling environment may be. The procedure itself, moreover, "can have a state which is preserved between calls to the procedure--and that state is independent of the calling environments."

Linden points out that a protected procedure will appear both as a subject and as an object, when represented in a protection matrix. A protected procedure is an object because there may be other subjects that have the right to call it. This right is represented in a protection matrix by the appearance of a special access right, such as the "enter" access right referred to earlier. A protected procedure also occurs as a subject in a protection matrix because, naturally, "it executes in its own protection domain."

Switching protection domains involves calling a protected procedure. The simplest case of domain switching is the one in which no access rights are passed as parameters in the call. The call takes place and execution begins in the protection domain of the called procedure, as long, of course, as the caller has the right to call this procedure in the first place. Return to the previous protection domain, i.e., the protection domain of the caller, is triggered by a return instruction in the executing called procedure.

This situation is illustrated in the protection matrix in Figure 2 [Lind76, p.417]. User A can call the editor, while executing in his own protection domain. He can also read or write files X and Y either from his own domain or by calling the editor, which is also allowed to read or write files X and Y. The user can use the dictionary, however, only by calling the editor, because the editor, but not the user himself, is allowed to read the dictionary.

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900



| OBJECTS / SUBJECTS | EDITOR | FILE X | FILE Y | DICTIONARY | |
|---|---|---|---|---|---|
| • • • | | | | | |
| USER A | ENTER | READ<br>WRITE | READ<br>WRITE | | |
| EDITOR | | READ<br>WRITE | READ<br>WRITE | READ | |
| • • | | | | | |

Figure 2: Simple Domain Switch

A domain switch is more complex if it involves the passing of access rights to objects as parameters "and if the protected procedure is to be reentrant." This kind of call to a protected procedure creates a new protection domain, i.e., a new row in the protection matrix. "The new protection domain contains both the permanent access rights of the protected procedure," defined by a template domain associated with the procedure, "and the access rights that are passed as parameters in the call."

This kind of situation is illustrated in Figure 3 [Lind76, p.418]. Figure 3a shows the User A's own basic domain and the template domain of the editor. User A has the same access rights as he has in Figure 2, but the editor is allowed only READ access to the dictionary. It cannot read or write files X or Y, as it can in Figure 2. If the user wants to use the editor to read file X, however, he can pass access rights for file X to the editor in the process of calling the editor. This results in the creation of a new protection domain, labeled "INSTANCE OF EDITOR" in Figure 3b, in which the editor does have READ access to file X. Linden notes that "other users may be editing other files using other instances of the same editor."

K. G. Walter [Walt75] presents what is, in effect[1], a formalization of Linden's account of security in the form of a model for mandatory security. Walter designs his model to satisfy the "design requirements...that there be no unauthorized disclosure of information and that, otherwise, unrestricted sharing of information be allowed." The model is based on the idea of restricting access to information by giving a specific classification for each piece of information and requiring a user to have the proper clearance in order to access the information.

[1]Calling Walter's characterization of security a formalization of Linden's is probably historically inaccurate, since Walter's account appeared a year and a half earlier than Linden's. This is the logical relation between the two theories, however, as we show in the text.

| objects / subjects | EDITOR | FILE X | FILE Y | DICTIONARY |
|---|---|---|---|---|
| • • • | | | | |
| USER A | ENTER | READ WRITE | READ WRITE | |
| EDITOR TEMPLATE | | | | READ |
| • • | | | | |

Figure 3a: Protection Matrix before Call to Editor

| objects / subjects | EDITOR | FILE X | FILE Y | DICTIONARY |
|---|---|---|---|---|
| • • • | | | | |
| USER A | ENTER | READ WRITE | READ WRITE | |
| EDITOR TEMPLATE | | | | READ |
| INSTANCE OF EDITOR | | READ WRITE | | READ |
| • • | | | | |

Figure 3b: Protection Matrix during Call to Editor

to access the information.

Formally, Walter describes his model as an 8-tuple

$$M_0 = (R, A, C, \theta, \mu, \trianglelefteq, Cls, Clr)$$

where

| | |
|---|---|
| R | is a set of repositories. |
| A | is a set of agents. |
| C | is a set of security classes. |
| $\theta \subseteq A \times R$ | is the "observe" relation. ($\underline{a}\ \theta\ \underline{r}$ means that agent $\underline{a}$ can observe the information stored in repository $\underline{r}$.) |
| $\mu \subseteq A \times R$ | is the "modify" relation. ($\underline{a}\ \mu\ \underline{r}$ means that agent $\underline{a}$ can modify the information stored in respository $\underline{r}$.) |
| $\trianglelefteq \subseteq C \times C$ | is a pre-ordering of the set of security classes. |
| CLS: $R \rightarrow C$ | is the "classification" function which associates a security class with each repository. (Informally Cls($\underline{r}$) will be referred to as the classification of repository $\underline{r}$.) |
| CLR: $A \rightarrow C$ | is the "clearance" function which associates a security class with each agent. (Here again Clr(a) will be referred to as the clearance of agent $\underline{a}$.) |

Walter's repositories correspond to Linden's objects, while his agents correspond to Linden's subjects. The observe and modify relations correspond to two general kinds of access right that can occur in a protection matrix. The security classes in $M_0$ correspond to Linden's small protection domains; it is they that determine which repositories

(objects) an agent (subject) can observe or modify (access). There is nothing in Walter's model that guarantees a null intersection of the classes of agents and repositories, so, as with Linden, it is quite possible for some (or all) of the entities involved to be both subjects and objects.

Walter imposes four axioms on his 8-tuple $M_0$ in order to prove his basic security theorem. The first two axioms state explicitly that the relation $\trianglelefteq$ provides a pre-ordering of the set C of security classes.

Axiom 1: For all $c \in C$, $c \trianglelefteq c$. ($\trianglelefteq$ is reflexive.)

Axiom 2: For all $c, d, e \in C$, $c \trianglelefteq d$ and $d \trianglelefteq e$ implies $c \trianglelefteq e$. ($\trianglelefteq$ is transitive.)

"The second two axioms govern, respectively, the acquisition and dissemination of information."

Axiom 3: For all $a \in A$ and $r \in R$, $a \ \theta \ r$ implies $Cls(r) \trianglelefteq Clr(a)$.

That is, if agent a can observe repository r, then the clearance of a must be greater than or equal to the classification of r).

Axiom 4: For all $a \in A$ and $r \in R$, $a \ \mu \ r$ implies $Clr(a) \trianglelefteq Cls(r)$.

That is, if an agent a can modify repository r, then the clearance of a is less than or equal to the classification of r. Agent a can modify only those repositories with equal or higher security class.)

Walter says that "for making comparisons it is sufficient to assume that the set of security classes is pre-ordered," (p. 286) but his earlier statement that "the classification system has a lattice structure" (p. 286), suggests that he really wants a partial ordering, since it is partial orderings that induce lattice structures. Formally, we include a third ordering

axiom to the effect that something cannot be both higher and lower in the ordering than something else, as follows:

> For all c, d $\varepsilon$ C, c $\underline{\lhd}$ d and d $\underline{\lhd}$ c implies c = d.

The basic security theorem states that "no information can ever be transferred to a repository in which it can be observed by an agent that does not have sufficient clearance to observe the source repository." Proving this theorem requires the introduction of a "transfer" relation $\tau \subseteq R \times R$, meaning that there is an agent that can transfer information from the first member of R to the second in a particular member of $\tau$. Formally, we say that $\underline{r}\ \tau\ \underline{s}$ for $\underline{r} \varepsilon R$, $\underline{s} \varepsilon R$, if and only if there is an $\underline{a} \varepsilon A$ such that $\underline{a}\ \Theta\ \underline{r}$ and $\underline{a}\ \mu\ \underline{s}$.

The basic security theorem itself requires the reflexive, transitive closure $\tau^*$ of $\tau$ and the notion of information transfer path. The relation $\underline{r}\ \tau^*\ \underline{s}$ means that "there is a finite sequence of repositories $\{\underline{r}_1\}$ such that $\underline{r} = \underline{r}_1$, $\underline{s} = \underline{r}_{n+1}$, and $\underline{r}_i\ \tau\ \underline{r}_{i+1}$ for all i, $1 \leq i \leq n$." In other words, $\underline{r}\ \tau^*\ \underline{s}$ if and only if information can eventually be passed from $\underline{r}$ to $\underline{s}$. We say that "there is an <u>information transfer path</u> from repository $\underline{r}$ to repository $\underline{s}$," if it is, in fact, the case that $\underline{r}\ \tau^*\ \underline{s}$.

Walter's basic security theorem can be stated formally in either of two ways, as follows:

> Theorem: For all r, s $\varepsilon$ R, if r $\tau^*$ s, then Cls(r) $\lhd$ Cls(s). In other words, if there is an information transfer path from repository $\underline{r}$ to repository $\underline{s}$ then Cls(r) $\underline{\lhd}$ Cls($\underline{s}$).
>
> Corollary: If $\underline{r}$ and $\underline{s}$ are repositories and the classification of $\underline{r}$ is not less than or equal to the classification of $\underline{s}$, then there is <u>no</u> information transfer path from $\underline{r}$ to $\underline{s}$.

What this theorem says is that if information flows from one repository to another, then the latter has a security class that is the same as or higher than the former; in other words, information can flow only upwards. Guaranteeing that, in a nutshell, is what the security problem is all about.

## 3. SPECIFICATION, IMPLEMENTATION, AND LEVELS OF ABSTRACTION

Walter does not stop with $M_0$, but also presents two other models, $M_1$, which is outlined in Figure 4, and $M_2$, which is too complicated simply to exhibit in a figure without further explanation. Walter describes the relationship that is supposed to exist between successive models in the sequence $M_0$, $M_1$, $M_2$ in terms of a "technique of structured modeling" (p. 288), in which successive "levels of modeling" are used to arrive at the full description of a system. He also uses the term "Structured Specification" (p. 285) to denote the approach to specification that results in models that are related in this way. Model $M_1$ "will satisfy the security requirements in $M_0$ plus further design requirements... These additional restrictions make the design more implementation specific" (p. 288) by representing the security system as "a file system structured as in a tree of arbitrary depth" and by providing "a mechanism for inter-agent communication which does not require accessing a shared file" (p. 290).

$M_2$ is a still "more specific security system model" (p. 290) involving "mechanisms which will be used as discretionary controls for access to files." Walter says that the definition of $M_0$ "has intuitive appeal, however, the way to apply $M_0$ to a complex operating system is far from obvious" (p. 293). As for $M_1$, "though still fairly general, this model is appropriate for a small class of machines. The next model, $M_2$, is applicable to few systems besides Multics," i.e., is getting very close to a description of (part of) an actual operating system, as implemented. "Eventually, some model (probably an $M_3$ or $M_4$) will closely resemble commands in the Multics System."

A general framework for understanding what Walter is trying to do is provided by the SRI systems model described by

$M_1$ = $(F, M, A, C, \rho_F, \sigma_F, \rho_M, \sigma_M, \trianglelefteq, \delta, Cls, Clr)$

WHERE:

| | |
|---|---|
| F | is a tree of files |
| M | is a set of mailboxes |
| A | is a set of agents |
| C | is a set of security classes |
| $\rho_F \subseteq A \times F$ | is the "retrieve information" relation. (a $\rho_F$ f means that agent a can retrieve information from file f.) |
| $\sigma_F \subseteq A \times F$ | is the "store information" relation. (a $\sigma_F$ f means that a can store information in f.) |
| $\rho_M \subseteq A \times M$ | is the "receive" relation. (a $\rho_M$ m means that agent a can receive information through mailbox m.) |
| $\sigma_M \subseteq A \times M$ | is the "send" relation. (a $\sigma_M$ m means that a can send information to m.) |
| $\trianglelefteq \subseteq C \times C$ | is a pre-ordering of the set of security class. |
| $\delta \subseteq F \times F$ | is the "dominate" relation on the set of files. (It defines the "tree" structure on the files.) |
| Cls: $F \cup M \rightarrow C$ | is the "classification" function for files and mailboxes |
| Clr: $A \rightarrow C$ | is the "clearance" function for agents. |

AXIOMS FOR $M_1$:

A1.1: For all $c \in C$, $c \trianglelefteq c$ ($\trianglelefteq$ is reflexive).

A1.2: For all c, d, e $\in$ C, $c \trianglelefteq d$ and $d \trianglelefteq e$ implies $c \trianglelefteq e$ ($\trianglelefteq$ is transitive).

A1.3: For all a $\in$ A and f $\in$ F, a $\rho_F$ f implies $Clr(f) \trianglelefteq Clr(a)$. (An agent can only "retrieve" information from a file with equal or lower classification).

Figure 4: Walter's "Tree Structured Directory Model - $M_1$"

A1.4: For all $a \in A$ and $m \in M$, $a\ \rho_M\ m$ implies $Cls(m) = Clr(a)$.
(An agent can only "receive" information through a mailbox with classification equal to its own clearance).

A1.5: For all $a \in A$ and $f \in F$, $a\ \sigma_F\ f$ implies $Clr(a) \trianglelefteq Cls(f)$.
(An agent can only "store" information in a file with equal or greater classification).

A1.6: For all $a \in A$ and $m \in M$, $a\ \sigma_M\ m$ implies $Clr(a) \trianglelefteq Cls(m)$.
(An agent can only "send" information through a mailbox with equal or greater classification).

A1.7: For all $f \in F$, $f\ \delta\ f$ ($\delta$ is reflexive).

A1.8: For all $f, g \in F$, $f\ \delta\ g$ and $g\ \delta\ f$ implies $f = g$.
($\delta$ is antisymmetric).

A1.9: For all $f, g, h \in F$, $f\ \delta\ g$ and $g\ \delta\ h$ implies $f\ \delta\ h$.
($\delta$ is transitive).

A1.10: For all $f, g, h \in F$, $g\ \delta\ f$ and $h\ \delta\ f$ implies $g\ \delta\ h$ or $h\ \delta\ f$ (sic).
($\delta$ has the "tree" property).

A1.11: For all $a \in A$, and $f, g \in F$, $a\ \rho_F\ g$ and $f\ \delta\ g$ implies $a\ \rho_F\ f$. (In order to retrieve information from a file, an agent must be able to retrieve from (i.e. search) every file which dominates it).

A1.12: For all $a \in A$, and $f, g \in F$, $a\ \sigma_F\ g$ and $f\ \delta\ g$ and $f \neq g$ implies $a\ \rho_F\ f$. (In order to store into a file, an agent must be able to retrieve from or search every file which strictly dominates it. This specifically allows an agent to store in a file from which it cannot retrieve; i.e., write-up is permitted.)

A1.13: For all $a \in A$, and $f, g \in F$, $a\ \sigma_F\ f$ and $f\ \delta\ g$ implies $a\ \sigma_F\ g$. (Since it is expected that attributes of a file will be maintained in a dominating file (directory), if an agent can store into a directory file and thus change attributes of an inferior file, then the agent must also be able to store into (modify) the inferior file).

A1.14: For all $f \in F$, there exists an $a \in A$ such that $a\ \sigma_F\ f$.
(There are no files which cannot be stored into (modified) by at least one agent).

Figure 4: Walter's "Tree Structured Directory Model - $M_1$" (con't)

Robinson [Robi75], [Robi77].[2] Robinson characterizes a system description in terms of "a sequence of ordered pairs $\{(P_0, M_0), (P_1, M_I), \ldots (P_n, M_n)\}$... called a hierarchically structured program" (p. 272) in which $P_i$ is a set of abstract programs that run on the abstract machine $M_i$. He notes that, in general, the pairs will occur in a tree structure, and that he assumes a linear ordering only in order to simplify the argument.

> Each program runs on a machine, but since the collection of machines forms a hierarchy, the primitive operations of a machine at some level are realized by a set of programs running on a machine at the next lower level (one program corresponding to each operation of the machine) (p. 272).

"The programs abstract from the implementation details of machines on which they run" and "the only information available to a program is the external behavior of the machine."

The general idea of this structuring is illustrated in Figure 5, in which "$M_0$ is the most primitive machine and can be viewed as the instruction set for a hardware machine or as a higher-order language" and in which "$P_n$ is the abstract program at the highest level, running on machine $M_n$." The direction of the arrows in the diagram represent the flow of implementation, in the sense that, "for all values of $i (0 \leq i < n)$, the set of abstract programs $P_i$ running on the abstract machine $M_i$ implements the abstract machine $M_{i+1}$," while itself running on abstract machine $M_i$. "The system as a whole is equivalent to some program P running on a machine M, where $M = M_0$ and $P_n$ is an abstraction of P."

Each of the abstract machines in Robinson's framework "can be described as a module of Parnas...in which both the internal state and the transformation rules are characterized

[2] The model description in [Robi75] differs somewhat from that of [Robi77]. We will quote the latter, unless otherwise noted.

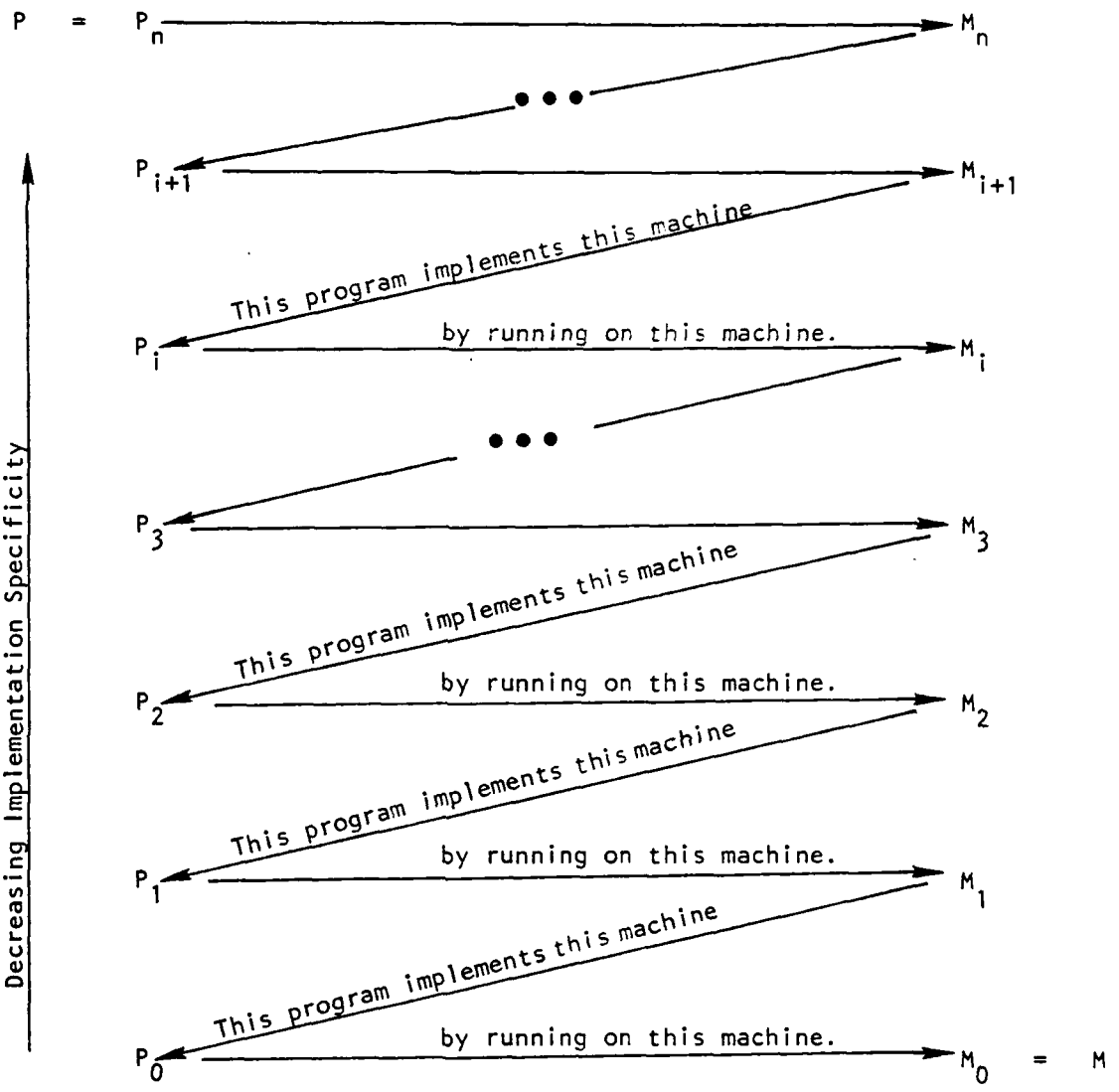



Figure 5

SRI Model of Program P to Run on Machine M

as functions of two types - V-functions (Value functions) and O-functions (Operation functions)." Each "program running on an abstract machine can be expressed as a sequence of calls to the functions that make up an abstract machine." A V-function is one that "returns a value when called; the set of possible V-function values of the module defines the state space (or abstract data structure) of the module." A module's state is denoted by a particular set of values for each V-function. O-functions describe state transformations by defining new values for V-functions. "A state transformation occurs when an O-function is called and is described as an assertion relating new values of V-functions to their values before the call." Such an assertion "is a predicate containing V-functions for which the predicate is true." It "specifies that, as a result of a call, the new state is one of some set of possible states; therefore the specification may be incomplete." The effect of this feature is that it "postpones binding of certain decisions until the abstract program is implemented or even until run-time." An example of an abstract machine characterized as a Parnas module specification is given in Figure 6 [Robi77, p.273].

Except for its reversed numbering scheme, it seems reasonably clear that the SRI framework we have just outlined corresponds more than roughly, in intent, to Walter's "technique of structured modeling" or "Structured Specification." Whereas Walter denotes his most abstract "level of modeling" by the number 0, with increasing numbers as we get closer to implementation, Robinson uses 0 to denote his least abstract "level of abstraction," with numbers increasing as we get further away from that level. The basic idea behind the separation of levels, however, is pretty much the same in both frameworks.

integer V-function: LENGTH

Comment: Returns the number of occupied positions in the register.
Initial value: LENGTH = 0
Exceptions: none

Integer V-function: CHAR(integer i)

Comment: Returns the value of the ith element of the register.
Initial value: ∀i(CHAR(i) = undefined)
Exceptions: I_OUT_OF_BOUNDS: i ≤ 0 ∨ i > LENGTH

O-function: INSERT(integer, i,j)

Comment: Inserts the value j after position i, moving subsequent values one position higher.
Exceptions:

I_OUT_OF_BOUNDS: i < 0 ∨ i > LENGTH

J_OUT_OF_BOUNDS: j < 0 ∨ j > 255

TOO_LONG: LENGTH ≥ 1000

Effects: LENGTH = 'LENGTH' + 1

∀k(CHAR(k) = if k ≤ i then 'CHAR'(k)
else if k = i + 1 then j
else 'CHAR'(k-1))

O-function: DELETE(integer i)

Comment: Deletes the ith element of the register, moving the subsequent values to fill in the gap.

Exceptions: I_OUT_OF_BOUNDS: i ≤ 0 ∨ i > LENGTH

Effects:

LENGTH = 'LENGTH' - 1

∀k(CHAR(k) = if k < i then 'CHAR'(k)
else 'CHAR'(k + 1))

Figure 6

Robinson's Register Module Specification

Walter describes the idea behind his methodology as follows:

> In many ways, Structured Specification is similar to Structured Programming; "levels of specification" are analogous to the "levels of abstraction" discussed in Structured Programming. However, in some sense, these concepts are orthogonal to each other. Structured Programming is a technique for evolving an orderly description of how a particular problem will be solved. Typically, it is a matter of filling in the "nitty-gritty" details of an algorithm which is well understood.
>
> Conversely, Structured Specification concentrates on evolving an orderly description of precisely what problem is to be solved. In addition, the various levels of specification provide a forum for discussing why the program is being designed in a particular way. (p. 285).

Differences in terminology aside (for example, Robinson's "levels of abstraction" would seem to be intended to correspond to Walter's "levels of specification," as well, perhaps, as to the "levels of abstraction" of structured programming), the aim of Robinson's methodology is the same.

Robinson, like Walter, is concerned with specification, not with implementation, except as an ultimate aim. Systems must eventually be implemented, of course, but this is not the point. He describes his methodology as one which "formally represents a program in terms of levels of abstraction, each level of which can be described by a self-contained non-procedural specification." (p. 271). The point is that a program is intended to be characterized in terms of what it is supposed to do (non-procedural), rather than in terms of how (procedural) it is supposed to do it, exactly as Walter says.

Robinson's characterization of a level of abstraction in terms of abstract machines is not a problem, because this involves only a choice of conceptualization and does not necessarily have to affect the formal methodology in an adverse way. A problem is created by the use of abstract programs, however, in the actual characterization of the abstract machines. A program is, by definition, a sequence of instructions, and so is intrinsically procedural. Indeed, Robinson characterizes "a program running on an abstract machine...as a sequence of calls to the functions that make up an abstract machine" (p. 272), as we have seen. As long as a systems framework uses abstract programs to characterize the functions of his primitive machines, we are automatically dealing with the how of those functions, rather than the what, i.e., with their implementation, rather than their specification.

We should note Robinson's assertion that "the Parnas specification language expresses state transformations in a non-procedural way... A Parnas module specification is a self-contained medium for defining an abstraction: V-functions are primitive, and O-functions are described solely in terms of V-functions and the constructs of the assertion language." What he means, presumably, is that, since the O-functions can be reduced to ("described solely in terms of") the V-functions and since the V-functions are primitive, i.e., not further reducible, there is nothing more that he has to do to characterize the module. Those functions (O-) which can be reduced have been reduced and those functions (V-) which have not been reduced need not be reduced, because they cannot be reduced. That, after all, is the meaning of "primitive." While it is true that the primitive elements of a system (any kind of system) cannot (or need not) be further reduced (decomposed, described, etc.) in terms of other elements of the system, however, it by no means follows that there is no need to characterize them at all.

Consider a simple case from plane Euclidean geometry. In that geometry, we can take the notions of point and line as primitives and notions like rectangle, triangle, and vertex as non-primitives that can be described in terms of the primitives. Thus point and line correspond to Robinson's V-functions, since he says these are primitive, while rectangle, triangle, and vertex correspond to his O-functions, since he says these are not primitive, but "are described solely in terms of V-functions." A rectangle or a triangle can be described (roughly, to avoid getting too technical and missing the main point) as a particular configuration of lines, and a vertex can be described as a point that is the intersection of two lines. Thus the non-primitives are described in terms of the primitives, exactly as Robinson wants.

The story does not end here, however. While reduction of geometric entities ends at the level of point and line (and perhaps other primitives, which we are ignoring for simplicity), point and line themselves are then characterized in terms of each other, i.e., in terms of their mutual interaction, by means of axioms. Something is a point or a line if and only if it behaves in accord with the axioms. The axioms of a geometry, in fact, are its most important part, because everything else about the geometry follows from them, once the appropriate definitions of non-primitive entities in terms of primitive ones are stated.

What this means in Robinson's case is that it is not enough simply to state that the V-functions are primitive and leave it at that. Looking carefully at Figure 6, we see that the only way that V-functions are characterized within the module is in terms of informal comments, in English, that tell us what the functions are supposed to do. The formalism, however, places no constraints on what these functions can do, except for giving them initial values and (perhaps) restricting their domains. Literally, any function that has these initial

values and these domains can serve as the LENGTH and CHAR functions in the module. Since this is too general for what Robinson intends, he is forced to narrow down the candidate functions for LENGTH and CHAR by characterizing them outside of the module in terms of abstract programs, which do spell out formally and, by definition, algorithmically the functions that he wants. This step, however, ipso facto removes us from the realm of specification and places us in that of implementation. In the process, we lose "the major advantages of Parnas specifications." namely, "that they abstract from the algorithms of implementation and are self-contained" (p. 272).

We see that Parnas' modules do not really characterize their functions completely, as they are supposed to. One of the underlying reasons for this problem is that Parnas tries to make his modules do too much. Parnas confuses the need to decompose a system into subsystems with the need to characterize in precise terms the kinds of objects the system deals with, proposing that both needs can be satisfied with his single notion of module.

In many places, Parnas talks about "dividing the system into modules " [Par72b, p. 1053] and "decomposing a system into modules," so it is clear that modules are intended to be the kind of thing into which systems are decomposed. With respect to the STACK module in Figure 7, however, he tells us that it is proposed as a definition of a kind of object:

> We propose that the definition of a stack shown in Example 1 should replace the usual pictures of implementations (e.g., the array with pointer or the linked list implementations). All that you need to know about a stack in order to use it is specified there. There are countless possible implementations (including a large number of sensible ones). The implementation should be free to vary without changing the using programs. If the using programs assume no more about a stack than is stated above, that will be true. (p. 332)

It follows from these facts that Parnas is decomposing systems into kinds of objects, but this is not the sort of result he really wants. It is this kind of inadequacy that leads Robinson to try to augment the Parnas methodology with things like abstract programs.

```
Function PUSH(a)
possible values:  none
integer:  a
effect:  call ERR1 if a > p2 ∨ a < 0 ∨ 'DEPTH' = p1
     else [VAL = a; DEPTH = 'DEPTH" + 1;]

Function POP
possible values:  none
parameters:  none
effect:  call ERR2 if 'DEPTH' = 0
     the sequence "PUSH(a); POP" has no net effect if no error
        calls occur.

Function VAL
possible values:  integer initial; value undefined
parameters:  none
effect:  error call if 'DEPTH' = 0

Function DEPTH
possible values:  integer; initial value 0
parameters:  none
effect:  none

p1 and p2 are parameters.  p1 is intended to represent the
     maximum depth of the stack and p2 the maximum width or
     maximum size for each item.
```

Figure 7

Parnas' Stack Module

## 4. HOS AS A GENERAL SYSTEMS THEORY

Like Linden and Walter, HOS recognizes that there are essentially two modes of existence in the world, that of *being* and that of *doing*, and that everything generally manifests both modes at once. A given thing can either *be* or *do* and, in general, will both *be* and *do* at the same time. This dichotomy reflects the related bifurcation between *being* and *becoming*. If there is something that is *doing*, then there is something (perhaps the same thing) that is *being done to*, and this latter thing is therefore *becoming*. Again, in general, anything that is *doing* is also *being done to* and so is itself *becoming*, as well as *being*.

This enables us to understand the important relationship between *constancy* and *change*. If we remove the front element from a queue, for example, we still have the *same* queue, with one element removed, but we also have a *different* queue, i.e., the one that differs from the original one in exactly that element. The queue can still *be* the *same* queue, even though it has *become* a *different* queue, and we are free to choose whichever of these aspects of the situation fits our needs for any particular problem. We can also say the queue has changed its *state*, stipulating that the queue itself has not changed, but then it is the *states* that are *being* or *becoming*, so the same dichotomy emerges again on a higher level of abstraction.

Linden expresses the distinction between *being* and *doing* in terms of his distinction between *objects* and *subjects*, as we have seen. Objects are things that *are done to*, i.e., they simply *are*, rather than *do*. Subjects, in contrast, are things which *do*, and the objects are precisely the things they *do to*. Walter expresses this dichotomy in terms of his distinction between *repositories* and *agents*, as we have also

seen. Agents are things which do, and repositories are things which are and which therefore are done to by the agents. As we have discussed, anything, in general, will both be and do, so anything is both an agent and a repository and both a subject and an object, as Linden, and presumably Walter, would agree.

While both Linden and Walter thus recognize this fundamental dichotomy in any system, there are serious defects in their formulations of this dichotomy. The problem with Linden's formulation is that it is not formal. All he tells us is that "a protection model views the computer as a set of active entities called subjects and a set of passive entities called objects" (p. 415), with no formal characterization of what these subject/object things or their properties are supposed to be. Such an omission is perfectly justifiable in the context of the general survey sort of article in which it occurs, but it must be corrected in a complete systems theory.

Walter's formulation is quite formal, but it falters in a different respect. A fully general systems theory should be capable of expression at the highest possible level of generality. Like Linden's account it should state things solely in terms of subjects and objects, i.e., things that do and things that are, at this highest level of generality, while permitting subcategorizations of these basic categories, e.g., procedure, protection domains, etc., at lower levels of generality. Walter's problem is that he conflates levels by including something not at all on a par with agents and repositories with respect to generality, i.e., security classes, on the highest level of generality of his systems theory. Again, within a sufficiently limited domain of interest, Walter's decision to lump the highly specific notion of security classes in with the completely general notions of agent and repository is excusable, but outside of such a domain,

it will place unnecessary restrictions on any system specified in accordance with the theory. A general systems theory should allow the introduction of lower-level notions like security classes, if they are needed, but it should not require them on its most general level, where only agents and repositories should reside.

HOS expresses the distinction between being and doing in terms of the familiar notions of data and function, and it does this in a completely formal way. Anything that can be can be represented as a member of a data type, and anything that can do can be represented as a function[3]. As we would expect from a correct formulation, anything that can be, i.e., a datum, can also do, by serving as input to a function, and anything that can do, i.e., a function, can also be, since functions themselves make up a data type.

For example, if datum x is mapped by functions $f_1$, $f_2$, $f_3$, $f_4$, $f_5$ onto data $y_1$, $y_2$, $y_3$, $y_4$, $y_5$, respectively, then x itself can be viewed as a function that maps the data $f_1$, $f_2$, $f_3$, $f_4$, $f_5$ onto $y_1$, $y_2$, $y_3$, $y_4$, $y_5$. Functions themselves can be data, in other words, and data can be functions, depending on the requirements of the particular problem we are working on. If FXY is the subset of data type FUNCTION whose members map data type X into data type Y, then X is the subset of FUNCTION that maps FXY into Y. Both interpretations are correct, in general, and which one we choose depends on what we need for a specific problem.

In our formulation, however, unlike Linden's, this reversability follows naturally from the nature of data and functions. We do not really have to say explicitly that subjects can also be objects and vice versa, because that fact follows automatically from our identification of subjects with functions and objects with data.

[3] [Ham76a] uses the term "function" in a more highly restricted sense and the term "operation" in the sense of our "function." For our present purposes, the distinction is unimportant, and we will use the two terms interchangeably.

Again in accordance with the fundamental dichotomy, although data and functions are distinct components of systems, they are at the same time inseparable from each other, because each is characterized formally in terms of the other. A function consists of an input data type, called its domain, an output data type, called its range, and a correspondence, called its mapping, between the members of its domain and those of its range; a function can be characterized, therefore, as an ordered triple (Domain, Range, Mapping), where the components are as we have just stated. A data type consists of a set of objects, called its members, and a set of functions, called its primitive operations, which are specified by giving their domains and ranges, at least one of which for each primitive operation must include the data type's own set of members, and a description of the way their mappings interact with one another and, perhaps, with those of other functions; a data type can thus also be characterized as an ordered triple, this time (Set, DR, Axioms), where Set is the set of its members, DR is a statement of the domains and ranges of its primitive operations, and Axioms is a description of the interactive behavior of the mappings of the primitive operations.

An example of an HOS data-type specification, namely, type STACK, is given in Figure 8, written in the HOS specification language AXES [Ham76a]. It is not difficult to see that this specification avoids all of the problems that we discussed in connection with Parnas' stack module in Figure 7. The specification in Figure 8 has absolutely nothing to do, by itself, with system decomposition. It is a definition of a kind of object, plain and simple, and thus serves exactly the kind of purpose it is suited to serve, rather than trying to overextend itself, as Parnas' module does. Furthermore, it is entirely self-contained, because the primitive operations are characterized in terms of each other, rather than being left dangling in the "module" to be rescued by abstract

```
DATA TYPE:  STACK;
PRIMITIVE OPERATIONS:
stack_1 = Push(stack_2, integer_1);
stack_1 = Pop(stack_2);
integer_1 = Top(stack_1);

AXIOMS:
     WHERE Newstack IS A CONSTANT STACK;
     WHERE s IS A STACK;
     WHERE i IS AN INTEGER:
Top(Newstack) = REJECT;
Top(Push(s,i)) = i;
Pop(Newstack) = REJECT;
Pop(Push(s,i)) = s;
END STACK;
```

Figure 8

HOS/AXES Data Type Stack

programs. Finally, it is absolutely implementation-free, because any implementation, whether made up of vacuum tubes, transistors, integrated circuits, magnetic bubbles, or ice-cream cones, will be a satisfactory implementation, as long as primitive operations can be defined in the implementation that behave in accordance with the axioms.

An interesting thing happens when we try to specify Walter's $M_0$ in terms of HOS data types. The first thing we notice about $M_0$ is that repositories are more basic than agents. An agent, in Walter's terms, is anything that can observe or modify a repository, while a repository is anything at all that can be partially ordered. Walter says that "associated with each repository is a security class which measures the relative sensitivity of the information stored within it." Since the only real function of the security class is to measure "relative sensitivity," it follows that their function could be accomplished just as well by partially ordering the repositories themselves. This enables us first to characterize the class of repositories as a data type independently of the class of agents and then to characterize the class of agents as a data type in terms of the data type REPOSITORY. It also enables us to eliminate the class of security classes altogether from our model by imposing our partial ordering directly on the data type REPOSITORY and assigning each agent a maximal repository it can observe and a minimal repository it can modify. This confirms our earlier observation that Walter is conflating levels of generality in his model. Security classes can be introduced as a data type at a lower level of generality, if they are really needed for a particular problem, or if they are simply desired for reasons of convenience or elegance, but they have no place on the highest level of generality of a general systems theory.

Figure 9 gives the HOS specification of data type REPOSITORY, written, as usual, in AXES. As just noted, the only primitive operation we need in this data type specification is the partial ordering Atmost, whose axioms are available with AXES and thus do not need to be stated explicitly.[4]

```
DATA TYPE:  REPOSITORY;
PRIMITIVE OPERATIONS:
   boolean = Atmost(repository1, repository2)
AXIOMS:
END REPOSITORY;
```

Figure 9

HOS specification of data type REPOSITORY

Note that whereas Walter treats his partial ordering as a general relation, i.e., as a general subset of C x C, or equivalently, a general set of ordered pairs $(C_1,C_2)$, we treat it as a function, i.e., a subset of REPOSITORY x REPOSITORY x BOOLEAN in which the first two components of each $(R_1,R_2,b)$ uniquely determine the third. The possibility of treating any relation as a function that maps into BOOLEAN is a general property of relations which HOS takes full advantage of. It enables us to integrate the treatment of relations that might not normally be viewed as functions into the general functional-decomposition framework of HOS and thus to see how such "non-functional" relations fit into the system as a whole of which they are a part.

[4]Equality is also needed, but this is provided in AXES itself for every data type. Atmost is not a universal operation, as Equality is, but is universally available, in that we can include it in any data type specification with whose axioms its own axioms are consistent. The axioms of Atmost are stated once and for all in AXES and thus need not be restated every time the operation is included among those of a particular data type. Once Atmost is included among the primitive operations of a particular data type, its axioms are automatically those that are stated for it in the theory. See [Cus77a] for discussion of these ideas.

Figure 10 gives the AXES specification for data type AGENT[5]. As noted earlier, there is one primitive operation, Observeclearance, that assigns to each agent a maximal repository it can observe and a second primitive operation, Modifyclearance, that assigns to each agent a minimal repository it can modify. The remaining two operations, Observes and Modifies, correspond to Walter's $\theta$ ("observe") and $\mu$ ("modify") relations, respectively, in the way discussed in the preceding paragraph.

```
DATA TYPE:  AGENT;
PRIMITIVE OPERATIONS:

  repository = Observeclearance(agent);
  repository = Modifyclearance(agent);
  boolean = Observes(agent,repository);
  boolean = Modifies(agent,repository);

AXIOMS:
  WHERE a IS AN AGENT
  WHERE r IS A REPOSITORY
(Observes(a,r) ⊃ Atmost(r,Observeclearance(a)) = True;
(Modifies(a,r) ⊃ Atmost(Modifyclearance(a),r)) = True;
Atmost(Observeclearance(a), Modifyclearance(a)) = True;

END AGENT;
```

Figure 10. HOS Specification of Data Type AGENT

The three axioms of data type AGENT together provide the effect of Walter's Axioms 3 and 4, without the use of "security classes." The first axiom says that if an agent can observe a repository, then that repository must be lower (but not necessarily strictly lower) in the partial ordering of repositories than the maximal repository the agent can observe. The axiom functions, in other words, as a mutual definition of "can observe" and "maximal observable repository" in terms of each other and the partial ordering, in the usual manner of HOS data-type axioms. The second axiom says that if an agent can modify a repository, then that repository must be higher (though perhaps not strictly higher) in the partial ordering of repositories than the minimal repository that the agent can modify. This functions, again, as a mutual definition of "can modify" and "minimal modifiable repository" in terms of each other and the partial ordering.

Given the first two axioms, the third axiom provides all of the effect of Walter's "security classes" by guaranteeing that the maximal observable

[5]The symbol "⊃" is a traditional infix symbol for material implication in formal logic and is used here in place of the AXES prefix operation symbol "Entails" [Ham76a]. It seems reasonable to use such traditional infix symbols as abbreviations for AXES prefix symbols, whenever this is convenient, and this convention is adopted explicitly in [Ham76a] and [Cus77a].

repository is always lower in the partial ordering than the minimal modifiable repository. This means that, for a given agent, the lattice of repositories can be divided into an "upper half" and a "lower half," such that the agent can observe only repositories in the lower half and modify only repositories in the upper half. This, however, is really the only purpose that security classes serve in $M_0$, so we really can dispense with them entirely, as we have done.

In Walter's terminology, we have reduced his 8-tuple

(R, A, C, $\theta$, $\mu$, $\trianglelefteq$, Cls, Clr)

to a 7-tuple

(REPOSITORY, Atmost, AGENT, Observes, Modifies, Observeclearance, Modifyclearance)

by showing that one of his data types is superfluous and that his primitive operations that map into that type can be replaced by different primitive operations which have the same effect but which have only the two remaining data types as domains and ranges. Whereas Walter's 8-tuple requires two special axioms, besides those for the partial ordering, which are intrinsic to AXES, but which Walter has to state, making a real total of five axioms for him, our 7-tuple requires only three explicitly stated axioms, as shown in Figure 5.

It should be noted that if we had tried to specify explicitly all three data types that Walter proposes, we would immediately have run into problems. Walter names his data types and describes how his operations (functions/relations) are supposed to work, but he does not explicitly specify either the operations or the types. His Axioms 3 and 4, for example, really express relationships between types, rather than defining characteristics of the individual types themselves. From the HOS point of view, this amounts to putting the cart before the horse, stating a relationship between two things before we have any idea at all what it is that is being related. From Walter's point of view, of course, this is perfectly legitimate, because, presumably, he views the situation as being

analogous to that of points and lines in plane geometry, which also are usually characterized not independently as data types, but in terms of each other. The advantage of our point of view is its complete generality. Identifying being things and doing things with data (types) and functions, respectively, enables us to specify any system at all in a principled way, without introducing any further kinds of entities. Walter's formulation of this distinction in terms of a mutual definition of repositories and agents, in contrast, still requires him to use functions (and relations, for that matter) to define his repositories and agents. In our framework, repositories and agents are data and functions, respectively, and that is the end of that.

We could have defined a type "SECURITY CLASS" in terms of the partial ordering, for example, but then we would have been unable to write axioms on the data types AGENT and REPOSITORY for the "primitive operations" CLS and CLR that map these types into that type, without introducing a host of other "primitive operations." Similarly, there would have been *no non-arbitrary way to decide* whether $\theta$ and $\mu$, which take both agents and repositories as input, should be "primitive operations" on AGENT or on REPOSITORY. By recognizing that the only function of "SECURITY CLASS" in $M_0$ is to provide an appropriate partial ordering for REPOSITORY, we can see that REPOSITORY is a more basic data type than AGENT and define the partial ordering directly on REPOSITORY, as we did. In other words, REPOSITORY is "SECURITY CLASS" at the level of generality at which $M_0$ is defined. Whether we call that single type "REPOSITORY" or "SECURITY CLASS" is, of course, entirely a matter of choice.

The other important function that Parnas tries to make his modules serve, i.e., system decomposition, is specified in HOS in terms of decomposition trees, also called control maps. Given a system that involves certain data types, the function the system performs can be decomposed into a tree structure whose nodes are functions and whose terminal nodes, in particular, are primitive operations of the data types,

where the collective effect of the functions at the terminal nodes is the same as that of the system as a whole. Such tree structures are not intended to provide definitions of kinds of objects, as Parnas' modules are, but represent system decompositions into subsystems, plain and simple. An example of such a decomposition tree, for the function $y = \frac{a+b}{c-d}$, is shown in Figure 11. The domain and range of the decomposed function can be determined by the typed variables that represent inputs and outputs and by the primitive operations that appear at the terminal nodes. The tree itself is precisely what gives the mapping of the decomposed function, by showing how that mapping gets accomplished in terms of the collective behavior of the independently characterized primitive operations.

The key to the usefulness of these decomposition trees lies in the six HOS axioms, listed in Figure 12. It is these axioms, in fact, and their consequences, of course, that make HOS HOS. While HOS can specify any system that can be specified, the specification must be in accordance with these axioms or the system may be incomplete or unreliable. Any software system, in particular, that is specified in accordance with these axioms is automatically guaranteed to be reliable, in the sense that no data or timing conflicts can ever occur [Ham76b]. Formally, the axioms tell us that a well-formed HOS tree is always equivalent to a tree in which every node is occupied by one of the three primitive control structures, shown in Figure 13. Abstract control structures, defined in terms of the primitives may also appear in well-formed trees, and, conversely, any control structure, i.e., configuration of parent and offspring nodes, can appear in a well-formed tree as long as it can itself be decomposed into the primitives.

Such an HOS tree can be interpreted either as decomposing a function into primitive operations or as building up a function out of primitive operations. Which interpretation we

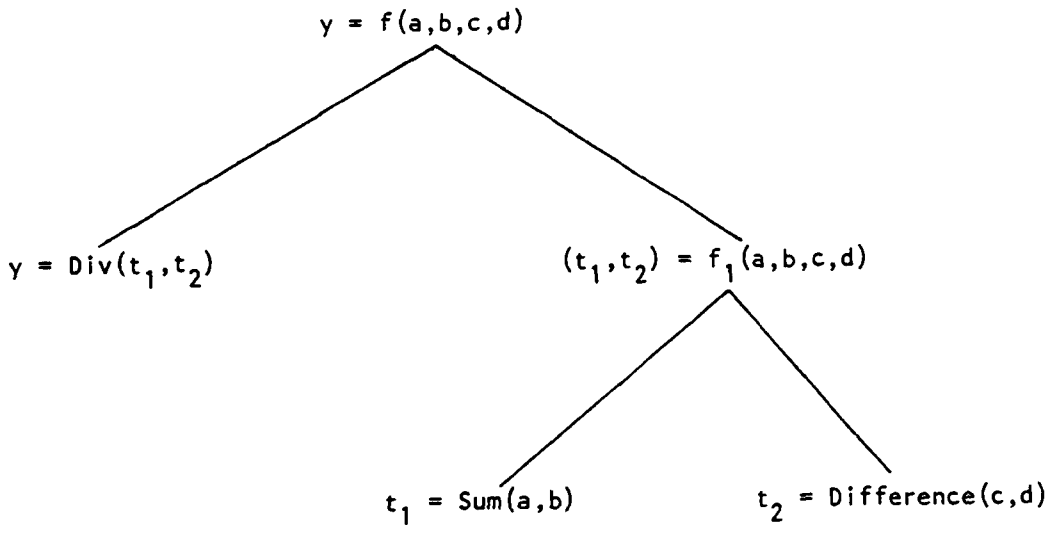



Figure 11

HOS Tree for Function $y = \frac{a+b}{c-d}$

DEFINITION: Invocation *provides for the ability to perform a function.*

*AXIOM 1:* A given module controls the invocation of the set of functions on its immediate, and only its immediate lower level.

DEFINITION: Responsibility *provides for the ability of a module to produce correct output values.*

*AXIOM 2:* A given module controls the responsibility for elements of its own and only its own output space.

DEFINITION: *An* output access right *provides for the ability to locate a variable, and once it is located, the ability to give a value to the located variable.*

*AXIOM 3:* A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate lower-level function.

DEFINITION: *An* input access right *provides for the ability to locate a variable, and once it is located, the ability to reference the value of that variable.*

*AXIOM 4:* A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate lower-level function.

DEFINITION: Rejection *provides for the ability to recognize an improper input element in that, if a given input element is not acceptable, null output is produced.*

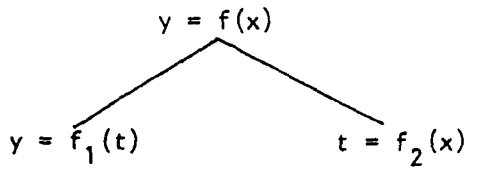
*AXIOM 5:* A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering *provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of the said elements precedes the other said element.*

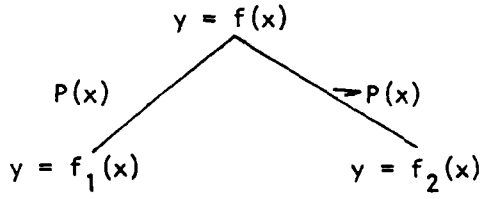
*AXIOM 6:* A given module controls the ordering of each tree for its immediate, and only its immediate, lower level.

Figure 12

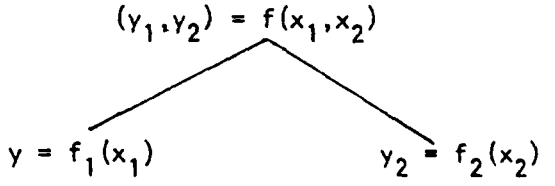
The Axioms of HOS




COMPOSITION




SET PARTITION




CLASS PARTITION

Figure 13

The Three Primitive Control Structures of HOS

choose for a particular tree depends, as usual, on the use we want to make of it. Under either interpretation of such a tree, however, what we end up with is a specification of the function at its root node that is genuinely non-procedural, i.e., non-algorithmic, and entirely free of implementation considerations. The tree provides a complete and explicit account of what functional mapping the function performs and how that mapping is collectively carried out on the types involved by their primitive operations. Everything is clearly spelled out in terms of the hierarchical organization of functional mappings, and this --no more, no less-- is exactly what we require of an adequate specification methodology. The need for abstract programs, i.e., (procedural) sequences of abstract calls to the primitive operations of abstract machines, is entirely eliminated. It follows that replacing each of Robinson's $P_i$'s with an HOS tree will make the problems we found in connection with his "abstract programs" disappear.

It is worth noting, at this point, that HOS does not distinguish at all between O-functions and V-functions, because, however important this distinction may be in particular implementations, it simply does not exist from the point of view of specification, i.e., on the highest level of generality. Functions are things that do, as opposed to be. Sorting out different kinds of functions is something we can do at lower levels of generality, but has no place as a requirement of the theory itself.

To illustrate this point again, suppose we have a register whose positions are filled with integers, as in the example of Figure 6 (a stack or queue would do just as well for our purposes; c.f. Figure 8 for data type stack and [Cus77b] for data type priority queue, for example). Obviously, there is a big difference between an implemented register and the integers it contains, and thus between changing the state of

the register and taking one of those integers as a value. From the point of view of specification, however, a register is every bit as much of an abstraction as an integer. The two abstractions differ, moreover, only in the interactive behavior of the primitive operations that are used to characterize their data types, as this behavior is specified in the axioms of the respective type. From the point of view of specification, therefore, changing the state of an implemented register amounts simply to producing a new abstract register as a value. If we take a register and remove its last element, for example, we get a new register that is identical to the original register except that it lacks the original register's last element. This may not be what happens in implementation, but it is the logic of the situation, and that is what specification is really all about[6].

As we observed earlier, Robinson supplements his "abstract machines" with "abstract programs" in order to do fully the two jobs that Parnas wants his modules to do. Robinson's "abstract programs" tell us what the functions really are that are intended to be characterized in the modules.

Robinson's intention can be successfully achieved by replacing each component of his framework with a corresponding component of HOS. Since his "abstract programs" serve as the characterizations of functions, we replace each of them with a decomposition tree. This relieves his "abstract machine" modules of the burden of serving as the units of system decomposition and leaves them free to serve as definitions of kinds of objects, which is what they would prefer to do anyway, as we have seen. We thus replace each of the "abstract machines" with a set of data-type specifications of HOS.

---

[6]Note that this is just another way of looking at what we said about queues in the second paragraph of this section.

Formally, then, we replace each of Robinson's ordered (P,M) pairs with an ordered pair (D,T), where D is a set of data types replacing the "abstract machine" M and T is a set of decomposition trees replacing the set of "abstract programs" P. Robinson's levels of abstraction gets replaced with a data level of HOS. For simplicity, we will assume that the data levels are linearly ordered, in order to preserve the analogy that we are developing with Robinson"s account of the SRI methodology, but, in fact, only a partial ordering is really necessary, as long as there is a maximal data level in the ordering that contains only one tree.

Higher data levels are related to lower data levels in that the composition trees of each data level decompose the primitive operations of the next higher data level in terms of the primitive operations of the lower data level[7]. For every primitive operation f of a member of $D_{i+1}$ ($i \geq 0$), in other words, there will be a decomposition tree in $T_i$ whose root is f and whose leaf nodes are primitive operations of a member of $D_i$. The primitive operations of the lowest-data level data types $D_0$ are the primitive operations of the system, because these are not decomposed at all, but are characterized only in terms of their axiomatic interaction. The $D_i$ thus play the role of Robinson's $M_i$ and the $T_i$ play the role of his $P_i$, as we said we want them to do, but avoiding any suggestion of implementation.

The simplest case, in which each $D_i$ contains a single data type and in which $T_n$ contains only one decomposed function f, corresponding to Robinson's single program P, is illustrated in Figure 14 which clearly reveals the parallel between the HOS

[7]We are restricting our discussion of HOS here somewhat, in order to maintain as close an analogy as possible with Robinson's framework. Later we will expand our account by discussing HOS in fuller generality.

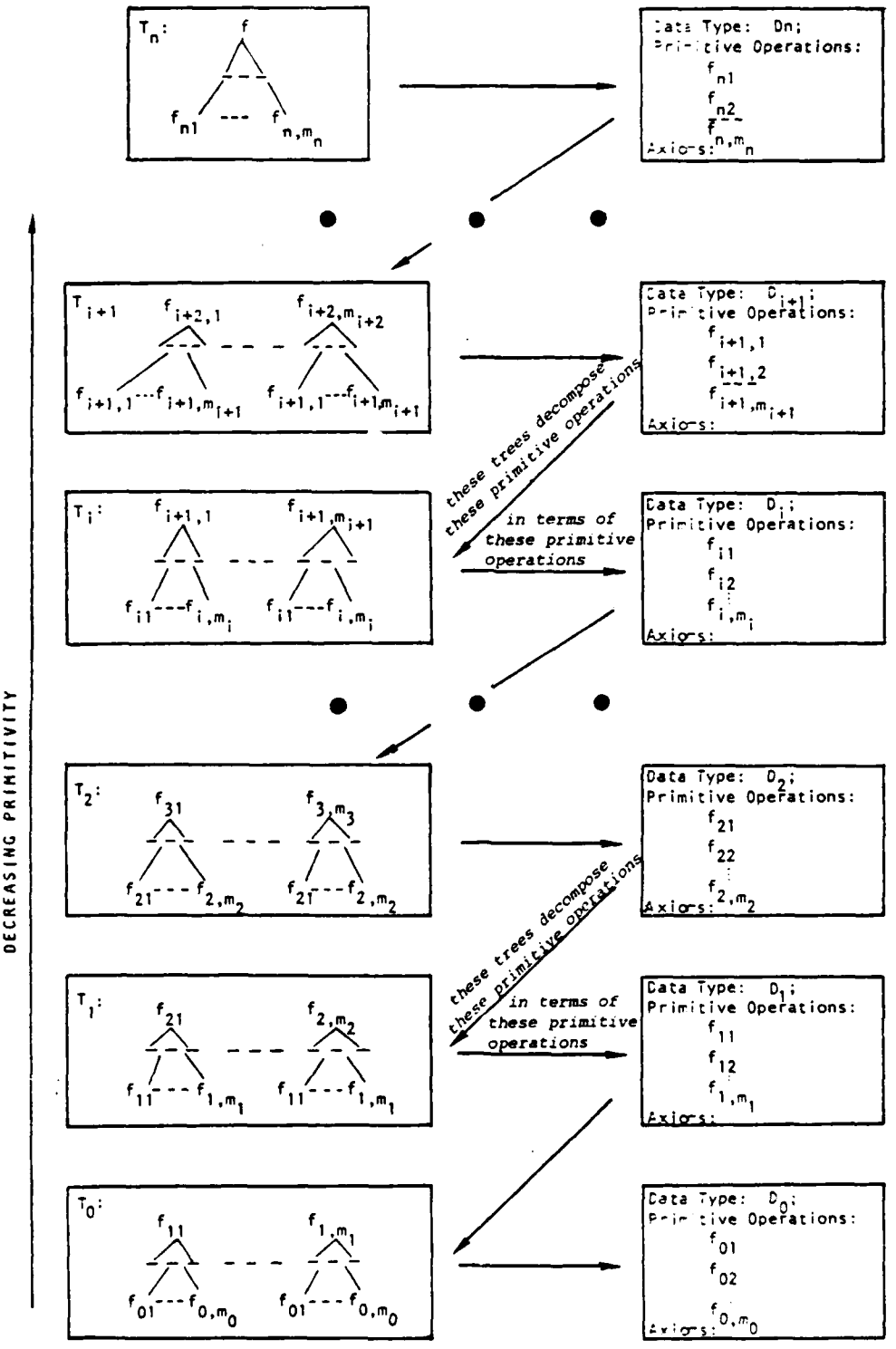



Figure 14

HOS Decomposition of Function f in terms of Primitive Operations of Data Type $D_0$

framework we have developed here and the SRI framework illustrated in Figure 5. The direction of the arrows in Figure 14 denotes flow of decomposition, however, rather than flow of implementation, as is the case in Figure 5. Everything in Figure 14 is strictly in the realm of specification and every subspecification ("module"), i.e., data types, trees, and data levels, is genuinely self-contained.

It is worth noticing at this point that Figure 14 suggests a way in which a relatively simple proof-of-correctness procedure might be developed for software specified in HOS. Robinson gives the following general account of how a proof-of-correctness procedure is supposed to work:

> The goal is to prove the correctness of a program P with respect to an input assertion, $\phi$, and an output assertion, $\psi$. Verification requires the insertion of inductive assertions $\{q_i\}$ into the program's flowchart, breaking the program into simple paths. Each simple path has one entry and one exit and between these a fixed number of executable statements. For each simple path, a formula called a verification condition (VC) must be stated and proved to be a theorem. The validity of all the VCs for a program is sufficient to demonstrate the partial correctness of a program--i.e. for all inputs satisfying the input assertion, the output assertion is satisfied if the program terminates. Termination can be proved by inductive assertions (usually different from those used to prove partial correctness) that bound the number of loop executions... (p. 274).

If we view Robinson's description in terms of Figure 14, we get the following general picture. What we need in proof-of-correctness is a set of intermediate points in the specification of a function, at which correctness assertions (verification conditions) are stated and can be proven. In Figure 14, such intermediate points appear to be provided automatically at each data level, where the axioms on data types can be viewed as assertions on the decompositions of higher-data level primitive

operations into lower-data level primitive operations. The input assertions are provided by a statement of what, in general, we intended the specified function to do. Spelling out this procedure in detail will require further work, but the general idea would seem to be clear.

## 5. HIGHER ORDER SOFTWARE IS SECURE SOFTWARE

Now we are in a position to return to our main topic of security. Given the parallels that we have developed between the SRI "specification" methodology and HOS, it would undoubtedly be useful to examine the SRI security model in view of these parallels and see whether we can shed any light on how that model can be tightened up, as we did for Walter's. There is good reason for not doing this, however. The SRI notion of security is very similar to Walter's, as we can see from the following description of that notion by Feiertag:

> In a multilevel secure system there is a predefined set of security levels. The security levels are composed of clearances (or classifications) and category sets, but the composition of the security levels is an unimportant detail for purposes of this discussion and will be largely ignored[8]. What is important is that the security levels are partially ordered by the relation "less than" represented by "<". Each process in a multilevel secure system is assigned a security level. The processes may invoke functions that change the state of the system and return values. Each function instantiation (i.e., a function with a particular set of argument values) is assigned a security level. A process may only invoke those instantiations of functions that have been assigned the security level of the process. A system is multilevel secure if and only if the behavior of a process at some given security level can be affected only by processes at a security level less than or equal to the given level. Stated in terms of functions, this says that the values returned by a function instantiation assigned some security level can be affected only by the invocation of function instantiations at lower or equal security levels. Stated in loose terms this means that information can flow only upward in the system from processes of lower security level to processes of higher security level. [Fei76, p. 1].

We already have enough at our disposal, however, to solve the security problem altogether, without trying to reexamine Feiertag's model in light of HOS. Doing the latter can thus be left simply as an interesting exercise for the reader.

[8] Like Feiertag, Walter also informally characterizes a "classification" as consisting of a "sensitivity level" and a "compartment," but, also like Feiertag, this distinction plays no real role in his formal security model. Note that here, too, a secure model is characterized as one in which information can flow only upward.

We arrived at our HOS model in Figure 14 by sticking fairly closely to Robinson's SRI model, as illustrated in Figure 5, and showing that each component of his model could be made completely free of implementation by replacing it with the corresponding HOS notion. What we found, essentially, was that the step from implementation to specification can indeed be made in somewhat the way Robinson wants, but only if we reformulate his notions in non-implementation terms. To capture successfully what Robinson is trying to express, we have to replace his "abstract machines" with HOS data-type specifications and his "abstract programs" with HOS function-decomposition trees.

In fact, however, HOS is considerably more general than the model in Figure 14. In particular, there is no reason for the relationship between the primitive operations of successive data levels to be related as directly as Figure 14 suggests. In the figure, the primitive operations of one data level are decomposed directly into the primitive operations of the next lower data level. In general, however, there can be intermediate operations on the lower data level that mediate this decomposition.

As we noted earlier, a data level of HOS is an ordered pair (D,T), where D is a set of abstract data types and T is a set of decomposition trees. We also said the data levels are linearly (or partially) ordered and that they are related in that the decomposition trees of each data level decompose the primitive operations of the next higher data level in terms of the primitive operations of the lower data level. In the most general case, however, the decomposition trees on one data level also use the primitive operations of that data level at their terminal nodes to define operations that do not appear as primitive operations of the next higher data level. In this case, there will be further decomposition trees between the data levels whose roots are primitive operations of higher

data levels and whose leaves are primitive or non-primitive operations of next-lower data levels.

To put the point a little differently, a data level of HOS, from the most abstract point of view, is nothing more than an ordered pair (D,T), where D is a collection of sets and T is a collection of mappings (mathematical functions). What makes such an ordered pair an HOS data level, is the kinds of constraints that are imposed on D and T by the HOS axioms (and their consequences). Every member of D is not only a set, but a set whose members behave towards each other in a way specified in an HOS data-type specification. Every member of T is not only a mapping, but a mapping that is decomposed in a well-formed HOS tree.

The mappings in T can represent completely general functions and do not have to be primitive operations on either their own or any other data level. If a mapping f is non-primitive on its own data level, then there is a decomposition tree that connects it to the primitive operations of that data level. Such a tree can be said to be horizontal, because it relates primitive and non-primitive operations on a single data level. There is also, however, a vertical tree that relates f to the primitive operations of the next higher data level. In this tree f is one of the leaves and the root is one of the primitive operations of the higher data level.

What we get instead of the arrows in Figure 14, in other words, is a retroflex step structure like the one in Figure 15. Each line segment in Figure 15 represents a set of decomposition trees, some of which are horizontal (on a data level) and some of which are vertical (between two data levels). The arrows point away from the root nodes and toward the leaf nodes of the trees they represent. Filled circles represent primitive operations of a data level, while filled squares

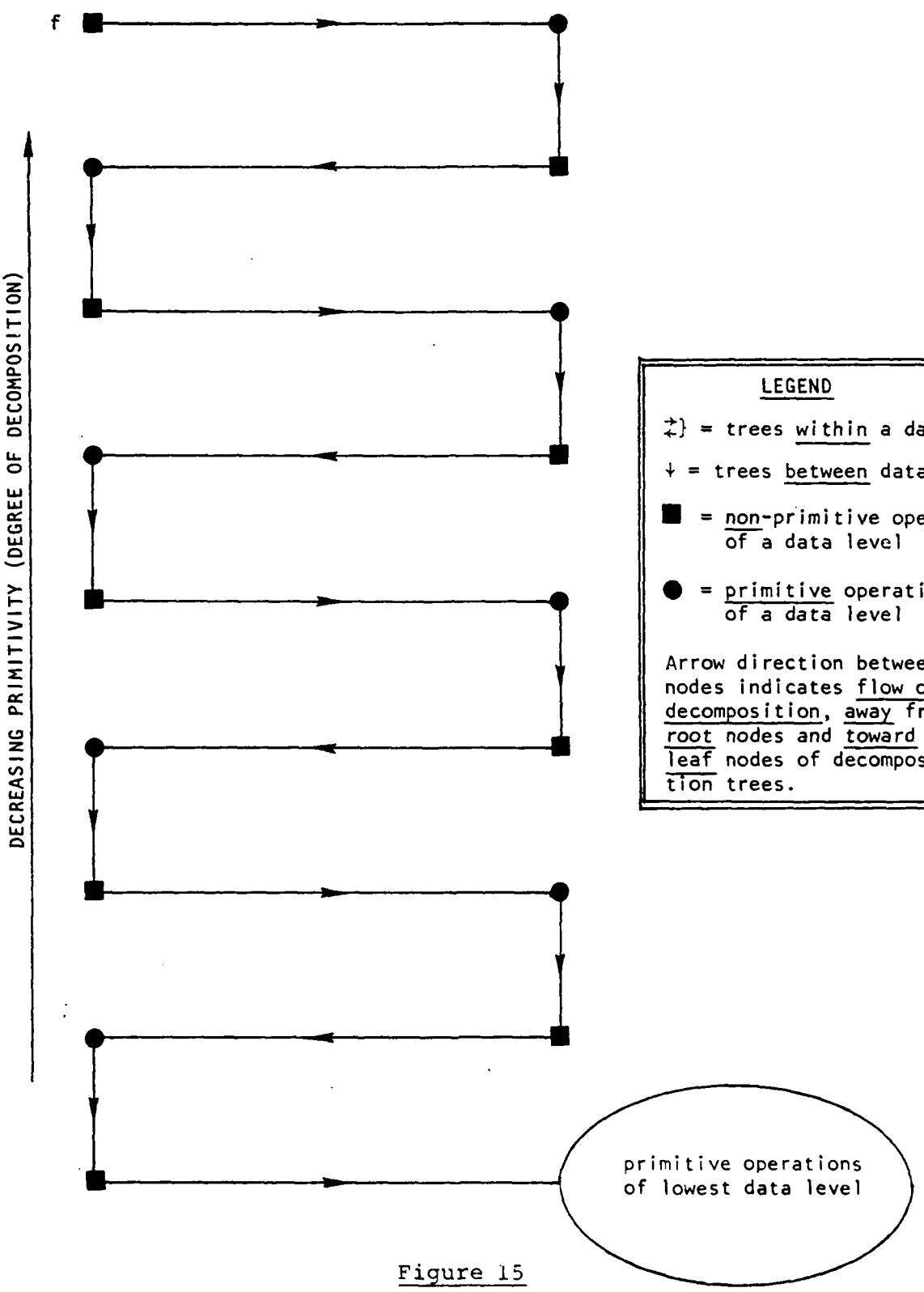



Figure 15

Retroflexed Step Structure of HOS Data Levels

denote non-primitive operations of a data level. Movement away from f produces increasingly decomposed operations (functions, mappings, etc.), i.e., an increasing degree of primitivity of the operations/functions involved. Movement towards f produces increasingly abstract or complex (decomposable) operations/functions, culminating in f itself.

In Figure 16, we elaborate this structure somewhat for a system with three data levels. As in Figure 15, filled circles in Figure 11 denote the primitive operations of a data level, while filled squares denote non-primitive operations of a data level. Open circles denote non-primitive operations that are needed in the intermediate levels of a decomposition tree[9]. These are described in terms of the three primitive control structures of HOS (composition, set partition, and class partition, as illustrated in Figure 13) or in terms of abstract control structures that are definable in terms of the three primitive control structures, as we mentioned in Section 4. Trees with solid branches are horizontal trees, which decompose non-primitive operations of a data level in terms of primitive operations of the same data level. Trees with dashed branches are vertical trees which decompose primitive operations of a data level in terms of non-primitive operations of the next lower data level. Note that primitivity of operations is a relative notion, defined with respect to the data level an operation is defined on.

Now we are ready to solve the security problem. Clearly, if we are not interested, for some reason, in the data-level structure of a particular f that has been decomposed as in Figures 15 and 16, then we can "fix" f in space, as it were, and "pull the rug out" from under the lowest data level, so that the filled nodes in the diagram act as pivots and the entire system stretches out into one gigantic tree structure, as in Figure 17. In conjunction with the HOS axioms, however,

[9] Note that HOS levels are defined relative to a controller, or parent module, whereas Robinson's are not. See [Ham76a,b,c].

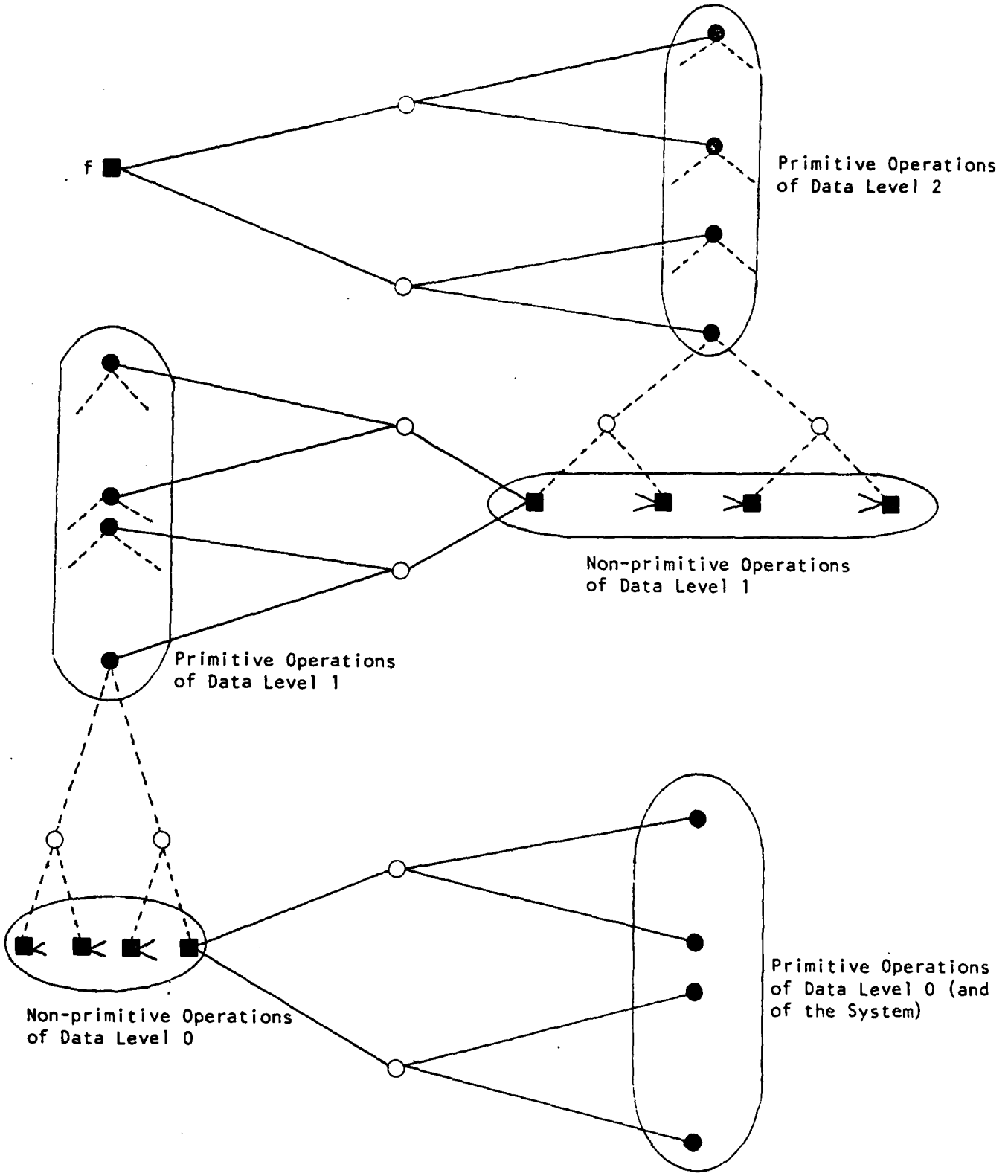



Figure 16

HOS Decomposition of Function f with Three Data Levels

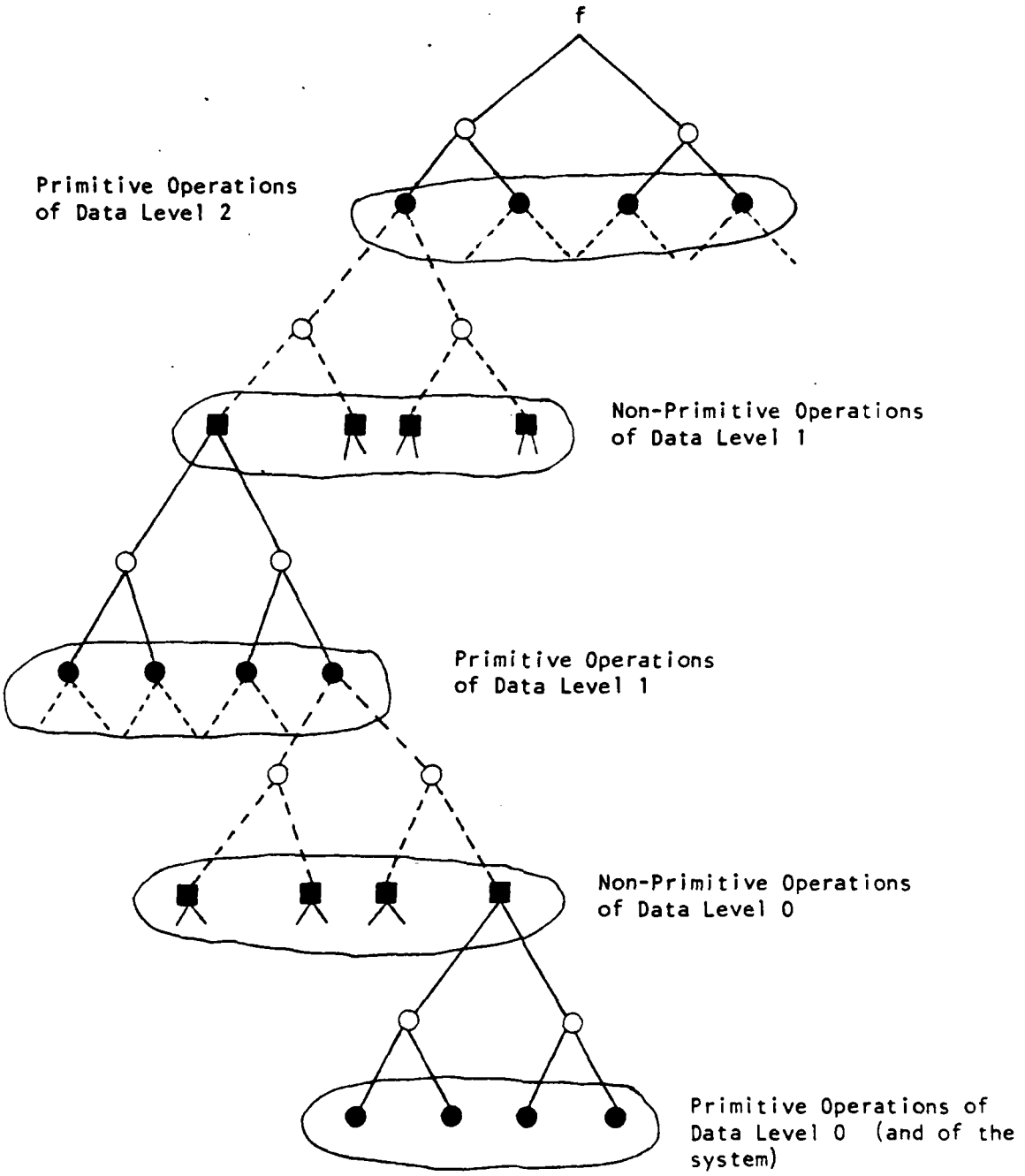



Figure 17

De-Retroflexed HOS Decomposition of Function f

this fact automatically provides us with the solution to the security problem.

Consider Axioms 3 and 4, in particular (Figure 12). These are the axioms that specify the access rights in an HOS system and would thus be expected to have something to do with security. Axiom 3 states that the access rights to the output of a function in a tree like that of Figure 17 are controlled by, and only by, its parent node ("module", in the axiom), i.e., the node immediately above it. Axiom 4, similarly, states that the access rights to the input of a function in such a tree is also controlled by and only by, its parent node. A given function can look at data only if its parent allows it to, and it must dispose of its results, again, only as its parent allows it to. It follows that the flow of control in an HOS system is always from the top down.

The flow of information, however, is always from the bottom up. A given node performs its function by having its offspring nodes, i.e., those on the immediately lower level, perform the function for it. This, in fact, is precisely what decomposition is really all about. Decomposing a function is just a formalized version of delegating responsibility. If someone can perform a task all by himself, then there is no point in delegating that task to subordinates. If responsibility is delegated, then he performs his task precisely by guaranteeing (via control) that his subordinates perform theirs. Formally, the offspring nodes look at the data that the parent allows them to (Axiom 4), perform their functions on that data as input, and then dispose of that data as the parent requires (Axiom 3), i.e., either by reporting it directly to the parent or by passing it on to an appropriate sibling. In particular, a given function has no idea what higher-level functions are doing. It just chugs along, turning input into output, disposing of that output as its parent tells it to. It is aware of what its offspring (or perhaps, siblings) are doing, however,

because that is precisely where it gets its input from in the first place.

The distinction between variables and values becomes very important here. Control is defined in terms of variables, but information is defined in terms of values. A node controls the access rights of its offspring to variables. The node tells the offspring what variables they can look at and what variables they must report back about. The offspring thus get their variables from the parent. This is the sense in which control flows downward. The parent node has no idea what the values of those variables are, however, until it gets those values from its offspring. The parent tells an offspring what variable to look at; then the offspring looks at the variable to find its value, operates on that value as input to change it into a value of an output variable, and then reports that value (either to a sibling or) back to the parent. Thus, while parents tell offspring what variables they can look at and assign, it is the offspring that tell the parents what the values of those variables are. It follows that information can flow only upward, precisely, in fact, because control flows downward, as stated in Axioms 3 and 4.

Our proof that information flows only upward in an HOS system required us to use the de-retroflexed tree in Figure 17, because the HOS axioms are stated in terms of trees (control maps), not in terms of retroflex trees, like the data-leveled structure in Figure 16. Since Figure 11 is functionally equivalent, however, to Figure 17, differing, in fact, only in its arrangement on the page, our proof of upward information flow is also valid for Figure 16.

The significance of this result cannot be overemphasized. As we saw in connection with Walter's $M_0$, a secure system is one in which the repositories (data) and agents (functions) are ordered in such a way, and the access rights of the agents

(functions) to the repositories (data) are assigned in such a way, that information can flow only upward in the ordering. What we have shown here, however, is that, if a system is specified in accordance with HOS, then its functions (agents) and data (repositories) are so ordered, and the access rights of the functions (agents) to the data (repositories) are so assigned, that information does always flow upward in the ordering. In other words, systems specified in HOS are automatically secure, without the need for any further paraphernalia to guarantee the security for us.

It follows that we have completely solved the security problem. If software is specified in HOS, then it is secure. It is that simple. Our proof of this fact also enables us to refine our discussion of HOS somewhat, and it would be worth while to pursue this opportunity a bit. We saw earlier that systems have a dual character in two distinct senses. On the one hand, a system is a function, since it performs a function, and it is also a datum in that it exists at all. On the other hand, a system consists of both data and functions and these two components are inseparable. What our proof of security makes clear, furthermore, is that each of these components has a dual character as well, and again, in two senses, when actually put together into a system.

A function in a system decomposition is a controller, because it controls the behavior of its offspring, in accordance with the axioms of HOS. It is also a performer, however, because it carries out the mapping of its parent. Every function plays both roles and the very fact that it plays one is the reason it must also play the other.[10] Data types also serve in two capacities in system decompositions. Every data type involved in a system decomposition provides both the input of one function and the output of another. "In" and "out" are as diametrically opposed as any two things can be, but,

[10] Primitive operations are also controllers (potentially), because we can always add a lower data level in which they are decomposed. Similarly, the highest-level function in a system is also a performer (potentially), because we can always use a system as a subsystem of some other system.

again, we cannot have one without the other.

Our components are also dual-natured in a second way. On the one hand, data have a constant aspect, as individual objects that can serve as inputs or outputs of functions, but, on the other hand, they have a variable aspect, because they exist as the members of data types. A given datum is an individual object itself, but it is also a representative member of a data type that can serve as a value of a variable of that type. This dichotomy enables functions to play a dual role in systems in a second sense as well. In Walter's terminology, a function can observe functions at a lower level of its decomposition tree by receiving data values from output variables of those functions and can modify functions at a higher level of its decomposition tree by providing data values to input variables of those functions.

On the one hand, therefore, functions act as agents, since they can observe lower-level functions and modify higher-level functions. What really gets observed and modified by these agents are the output variables of the respective functions with the modification occurring via the use of the input variables, so it is the output variables that serve as the repositories of the system. On the other hand, the input variables also function as agents because it is they that give the relevant values to their functions to use in modifying the values of the output variables. In general, in other words, it is the complete functions themselves--mappings, domains, and ranges, with the latter two represented by variables--that act both as agents and repositories in an HOS system.

It follows that we not only do not have to distinguish between repositories and security classes, as we saw earlier, but we do not really even have to distinguish between repositories and agents either. Since a function all by itself already

has a dual character, being made up of a mapping and two data types, functions themselves can play both roles. When functions occur in a tree, they can observe and modify other functions and they can also be observed and modified by other functions. Since they occur in a tree structure in any system, the functions themselves, and therefore the "agents" and "repositories," which the functions, are, are partially ordered, (and thus also pre-ordered), just as Walter wants them to be. A function in a system is both an agent and a repository and, since it occurs in a tree structure, can also serve as its own security class. This is about as cozy an arrangement as we could possibly want and, as we have seen quite clearly, it is absolutely secure.

## 6. SOFTWARE, SYSTEMS, SEMANTICS, AND BEYOND...

In most real scientific breakthroughs, the applicability, usefulness, and explanatory power of a new theory goes well beyond the restricted kind of problem it was originally intended to solve. All such developments clearly exhibit the contradictory aspects of similarity and difference, of continuity and change. Major breakthroughs always bear strong similarities to existing theories, but differ from them in key respects whose logical implications turn out to make all the difference.

These sorts of characteristic are clearly evident in the case of HOS as an approach to systems theory. We have seen that, while HOS was originally developed for the specification of reliable software, it turns out to provide automatically the solution to the security problem as well. Many HOS concepts look very much like the notions contained in other theories. Very close examination reveals, however, that HOS differs from other formulations in precisely the ways that are required by the problem the theories are trying to solve. What results is a completely adequate methodology for the specification of software systems that are reliable and secure.

In fact, what results is much more than that. HOS seems capable of providing insight into problems that are well beyond its intended field of software engineering. There is nothing in HOS, in other words, that requires us to restrict its use to specifying software systems. As a general systems theory, HOS can be fruitfully applied in any field in which systems can be seen to be playing a role. George Miller has suggested to us (personal communication) that HOS control hierarchies might be useful in accounting for the behavior induced by operant conditioning, and we have ourselves been investigating its usefulness as a tool

in analyzing natural language. The HOS distinction between data and functions, for example, can be interpreted as providing a semantic model, in the sense of Wilson [Wil76], [Mil176]. Wilson's own semantic model, illustrated in Figure 18, is considerably less general, recognizing seven modes of existence, which he calls "concept types": objects, properties, relationships, events, actions, procedures, and sets. Such a model would certainly be useful for many purposes, but its limited generality cannot help but conceal significant generalizations that might help to simplify specific system specifications and suggest alternate implementations. Wilson's model represents a number of possible implementations of the HOS model at a lower level of generality. His "object classes," "property classes," "relations/attributes," and "sets," for example, could represent a particular selection of data types, with "events," "actions," and "procedures" corresponding to different classes of functions. The former, after all, represent things that are (be), while the latter represent things that do.

The data/function dichotomy could be distributed among Wilson's seven "concept types" in other ways as well, but the question that immediately strikes one on first coming across his model is that of why he chooses these seven in the first place. The problem with Wilson's semantic model, in other words, as a general systems (or semantic) theory, is that it is essentially ad hoc and, therefore, of limited generality. Actions certainly constitute events, for example, so they could be subsumed under them. Reversing direction, we could elaborate actions further, distinguishing them into transitive and intransitive actions, perhaps. Properties, similarly, could be taken to be one-place relationships, as they often are. The point is that Wilson's theory provides no natural mechanism with which to make the plethora of such decisions that might arise in specific cases of system design. The number of "concept types" is stipulated to be seven, in the theory itself, and that is that. HOS, in contrast, distinguishes only between

CONCEPT TYPES

| Instances | Classes (Abstracted from Instances) |
|---|---|
| objects | object classes |
| properties | property classes |
| relationships | relations/attributes |
| events | event classes |
| actions | action classes |
| procedures | procedure classes |
| sets | set classes |

CERTAIN KEY RELATIONSHIPS BETWEEN CONCEPTS

INSTANCE of/CLASS of

SUBCLASS of/SUPERCLASS of

COMPONENT of SUPERCOMPOSITE of (PART/WHOLE)

MEMBER of/SET MEMBERSHIP of

SUBSET of/SUPERSET of

Figure 18. Wilson's Semantic Model [Wil76, p. 7]

data and functions, while permitting any instance of either to be also an instance of the other. Given this generality, we can begin restricting things anyway we like for any particular problem: three special data types and four classes of functions, two special data types and six classes of functions, ten special data types and one class of functions, etc. Once we let ourselves get more concrete than simply being versus doing, in other words, there is no clear general criterion for what our categories, modes of existence, or concept types should be, because each application or class of applications will favor a different choice. An adequate general systems theory must be formulated at the highest level of generality, so that no possibly desired choice of implementation will be ruled out, or made intrinsically more difficult, ahead of time.

By distinguishing only between data types and functions, in other words, HOS lets each particular more or less concrete application select exactly the specific data types and functions it needs, rather than arbitrarily stopping the theory short at a lower level of generality, and possibly ruling out the optimal choice of data types and functions for a particular application. The point here is not that Wilson's semantic model is wrong, but that, unlike HOS, it is not fully general, and, therefore, not fully adequate.

One final point remains to be made before we close. The reason that system specification has been such a difficult thing to figure out is that, as we have seen, a system is an intrinsically contradictory object. On the one hand, a system is a single object; on the other hand, it is made up of many different objects. On the one hand, a system performs a function on objects; on the other hand, it is an object on which functions can be performed. On the one hand, a system consists of two distinct kinds of objects, functions and data; on the other hand, functions can be data and data can be

functions, so only one sort of thing is really involved. A datum can be an input to a function, but it can be so only by being an output to a function, and vice versa. A function controls other functions, but it also gets controlled by another function. A datum is an individual object, with a constant aspect, and also a representative of a data type, with a variable aspect. Functions are defined in terms of variables, i.e., representatives of data types, but operate on constants, i.e., individual data, and so on.

Given all these twists and turns on the road to specification, it is not surprising that so many have lost their way. The uniqueness and power of HOS consists precisely in the fact that it manages to resolve all of these contradictions in one fell swoop and makes them comprehensible.

## REFERENCES


[Cus77a] Cushing, S. "A note on equality in AXES data-type specifications" (in preparation).

[Cus77b] Cushing, S. and Heath, W. "ARO operating system," ARO Memo #1. Cambridge, MA: Higher Order Software, Inc. (hereafter cited as HOS, Inc.), April 7, 1977.

[Ham76a] Hamilton, M. and Zeldin, S. "AXES syntax description." TR-4. HOS, Inc., December 1976.

[Ham76b] Hamilton, M. and Zeldin, S. "Higher order software--a methodology for defining software." IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976.

[Ham76c] Hamilton, M. and Zeldin, S. "Integrated software development system/higher order software conceptual description." Version 1. HOS, Inc., November 1976.

[Ham77] Hamilton, M. and Zeldin, S. "The manager as an abstract systems engineer." TR-5. HOS, Inc., June 1977. (To be presented at the COMPCON 77 Fall Conference, conducted by the IEEE Computer Society, Washington D.C., September 1977.

[Lind76] Linden, T. A. "Operating system structures to support security and reliable software." ACM Computing Surveys, VIII, 4. December 1976, pp. 409-445.

[Mill76] Mills, H. D. and Wilson, M. C. "An introduction to the information automat." Gaithersburg, MD: IBM, May 7, 1976.

[Par72a] Parnas, D. L. "A technique for software module specification with examples." Communications of the ACM, XV, 5. May 1972, pp. 330-336.

[Par72b] Parnas, D. L. "On the criteria to be used in decomposing systems into modules." Communications of the ACM, XV, 12. December 1972, pp. 1053-1058.

[Robi75] Robinson, L., et al. "On attaining reliable software for a secure operating system." Proceedings, International Conference on Reliable Software, Los Angeles. April 21-23, 1975.

[Robi77] Robinson, L., et al. "Proof techniques for hierarchically structured programs." Communications of the ACM, XX, 4. April 1977, pp. 271-283.

[Walt75] Walter, W. C., et al. "Structured specification of a security kernel." Proceedings, International Conference on Reliable Software, Los Angeles. April 21-23, 1975.

[Wil76] Wilson, M. L. "The information automat approach to design and implementation of computer-based systems." Gaithersburg, MD: IBM, April 1976.